High-performance submodular function minimization

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High-Performance Submodular Function Minimization

Master Thesis

Author:
Giovanni Azua Garcia
09-934-415

Supervisors:
Prof. Dr. Markus Püschel
Prof. Dr. Andreas Krause

Department of Computer Science ETH Zürich

Institute:	Institute for Computer Systems
Group:	Advanced Computing Laboratory
Institute:	Chair of Information Science and Engineering
Group:	Machine Learning and Adaptive Systems

September 20, 2012

Abstract

Combinatorial optimization is a branch of mathematical optimization that has important applications in many fields, including artificial intelligence, machine learning, computer vision, mathematics, auction theory, and game theory. The submodularity function property frequently appears in many combinatorial systems and enables efficiently finding solutions to these problems that would otherwise be intractable. Submodularity is in many ways a discrete analogue of convexity [45] and the combinatorial structure it provides, allows the efficient search for a near-optimal solution in strongly polynomial time [25] [38].

Submodular functions have received much interest in recent years and to date the practically fastest algorithm of choice for submodular function minimization (SFM) is the Minimum-Norm-Point (MNP) implementation by Fujishige-Wolfe [13]. In the present work and project we bring together ideas from interdisciplinary fields, including high-performance computing, computer systems, optimization, algorithms and software engineering to deliver the fastest and most robust practical algorithm implementation for general submodular function minimization. We successfully delivered a high-performance submodular function minimization (HPSFO) algorithm that reaches up to 90% of Vector peak performance, offers parallel speed up and outperforms all the existing tested implementations by not one but several orders of magnitude and for the three workload applications we implemented and that scales reasonably well with respect to the problem sizes. We tested our HPSFO with the following three workload applications: Minimum Graph Cut, Log Determinant and Text Corpus Selection in the context of Automatic Speech Recognition (ASR). Furthermore, this work provides a solid extensible software framework featuring a design that allows accommodating with ease new submodular function optimization algorithms and submodular applications.

Acknowledgements

I would like to first dedicate this Thesis to the memory of my grandfather Ramon Garcia who inspired my curiosity. I would like to thank Elena for taking full care of our small daughter Isabella while I have been busy working and writing this Master Thesis.

I would like to give special thank to my supervisors Prof. Markus Püschel and Prof. Andreas Krause, for their support and patience answering my many questions. I would also like to give a special thank to my ex-project team Zaheer Chothia and Andrei Frunza who supported this further work by dedicating time to discuss the new ideas and answer questions related to different areas of the project we initially worked on. I would like to particularly thank Zaheer Chothia for his contributions, e.g., tweaking CMake to compile and generate a Matlab MEX library.

I would like to thank Prof. Daniel Kressner for taking the time to answer multiple questions related to his work on updating an existing QR decomposition. I would also like to thank Prof. Julien Langou for answering many of my questions related to the use of the LAPACK library.

I would like to thank specially the MKL Intel team for their fantastic support and for answering many of my questions via their discussion Forum.

I would like to thank Change Vision, Inc. for providing a free version of Astah Professional [1] to create high-quality class diagrams.

Finally, I would like to thank Gonzalo Medina for answering my questions on TikZ related to building perfect diagrams, e.g., to perfectly represent a blocked Givens rotation stencil.

[1] `http://astah.net/`

Contents

Chapter 1

Introduction

Combinatorial optimization is a branch of mathematical optimization that has important applications in many fields, including artificial intelligence, machine learning, computer vision, mathematics, auction theory, and game theory. In the combinatorial optimization problem considered in this thesis we are given a *ground set* V, and we wish to find the *optimal subset* $X \subseteq V$ that minimizes (or maximizes) a given function $f : 2^V \to \mathbb{R}$; 2^V represents the family of all possible subsets of V. A naive brute force algorithm would evaluate the function f on all 2^n subsets of V and take, e.g. the minimum, but this is in general NP-hard. The submodularity property 1.1 frequently appears in many combinatorial problems and can dramatically reduce the complexity. Formally, a function f is submodular, if:

$$f(A) + f(B) \geq f(A \cap B) + f(A \cup B),\ \forall A, B \subseteq V. \tag{1.1}$$

An equivalent and more intuitive definition of submodularity states that adding more elements to a set brings *diminishing returns*:

$$f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B), \forall A \subseteq B \subset V. \tag{1.2}$$

Submodularity is in many ways a discrete analogue of convexity [45] and the combinatorial structure it provides, allows the efficient search for a near-optimal solution in strongly polynomial time [25, 38]. An example is, the problem of Text Corpora Selection, where we are given a ground set of utterances or expressions and related vocabulary and we wish to find the optimal subset of utterances that contains the maximum amount of information with minimum vocabulary size. This application is described in detail in the work of of Hui Lin and Jeff Bilmes [33] and this problem can be cast as a submodular function minimization (SFM) problem with submodular function defined in equation 1.3. Minimizing this objective function achieves our wish first by maximizing the amount of information in our selected subset X with the term $w(X^c)$ that effectively penalizes the function by the amount of information in the complement set $X^c = \{V \backslash X\}$; and secondly by penalizing the function with the vocabulary size defined by the term $\Gamma(X)$ (Hui Lin and Jeff Bilmes [33]):

$$f(X) = w(X^c) + \lambda\Gamma(X) \tag{1.3}$$

Submodular functions have appeared in many applications in recent years and to date the practically fastest algorithm of choice for general submodular function minimization (SFM) is the Minimum-Norm-Point (MNP) algorithm and implementation by Fujishige-Wolfe [13]. However, the implementation has multiple limitations: e.g., its applicability is limited due to poor scalability with respect to problem sizes and it is difficult to adapt or reuse in different application contexts. In this thesis we applied key performance improvements to the Fujishige-Wolfe MNP algorithm and delivered a high-performance implementation HPSFO that outperforms the state-of-the-art implementations and for all test applications as we will discuss in the Experimental results chapter 6. The runtime optimizations we implemented included both, algorithmic improvements that reduce the asymptotic runtime, and system level optimizations that map the algorithm efficiently to multicore processor architectures. Furthermore, we implemented three workload applications: Minimum Graph Cut, Log Determinant and Text Corpus Selection (ASR) to test and compare our HPSFO against all other available implementations.

1.1 Contributions

In this work we provided a contribution to the combinatorial optimization and machine learning community by delivering the fastest to date practical algorithm implementation for general submodular function minimization. We have applied Krause's concept of *incremental updates* [30] of the EO function evaluation and lowered its algorithmic complexity for all applications we tested. To illustrate Krause's incremental update lets take as example the submodular function suggested by Satoru Iwata (Fujishige-Isotani [13]):

$$f(X) = |X||X^c| - \sum_{j \in X}(5j - 2n) \tag{1.4}$$

where $X \subseteq V$, $X^c = \{V \setminus X\}$ and $V = \{1, 2, \ldots, n\}$. If we keep a context associated with the function that conveniently "remembers" the previous evaluation $f(X)$ then we can effectively lower Iwata EO evaluation complexity from $O(n)$ to $O(1)$, i.e., constant. The following demonstrate how to do so if we have already computed $f(X)$ and we wish to re-evaluate Iwata EO after adding a new element k to subset X:

$$f(X \cup \{k\}) = \underbrace{f(X)}_{\text{Saved in context}} - |X||X^c| + |X \cup \{k\}||V \setminus X \cup \{k\}| - (5k - 2n) \quad (1.5)$$

Furthermore, we implemented a fast high-performance version of the MNP algorithm based on Fujishige-Wolfe that outperforms all other SFM implementations by more than one order of magnitude. In addition to the performance improvements we also designed a solid software framework that features easy maintainability and extensibility for both new submodular optimization algorithms and new applications plus built-in integration with Matlab and a “zero-dependency” API for easy integration with other platforms, e.g. Java via JNI [1].

The contributions of this thesis are:

1. Explore and develop an algorithmic improvement based on *incremental update* of the evaluation oracle (EO) function. We extended and integrated into our design the idea from Krause of EO evaluation with incremental updates that reduces the evaluation complexity and therefore runtime. We successfully applied this idea to all applications and lowered the complexity (and cost) of the Log Determinant application from $O(n^3)$ to $O(n^2)$, lowered the complexity of the Minimum Cut from $O(n^3)$ to $O(n)$ and lowered the complexity of the Corpus Text Selection from $O(n^2 \log n)$ to $O(n)$.

2. Deliver a general SFM algorithm implementation optimized for locality, vector extensions, and multi-threading. We cast Krause and Fujishige-Wolfe implementations in terms of our high-performance foundation that builds on top of Intel MKL automatically gaining most of the aforementioned optimizations in a cross-platform portable fashion. We also identified key performance bottlenecks of the MNP and delivered tailored handcrafted performance solutions, e.g., Register blocking and loop unrolling Givens rotations. Our final HPSFO implementation reaches up to 90% of vector peak performance and offers parallel speed up but parallelism starts to pay off for bigger problem sizes.

[1]`http://docs.oracle.com/javase/6/docs/technotes/guides/jni/`

3. Provide a submodularity framework and implementation that is flexible, i.e., can be easily instantiated to various submodular function minimization problems. We combined the best of two seemingly orthogonal computer science fields: software engineering and high-performance computing; to build a robust, intuitive and extensible software framework that features prescriptive extensibility and allows accommodating and combining with ease multiple algorithms and applications.

4. Design a submodularity API that is reusable and portable for use in C, C++ and Matlab. Provide integration possibilities for Matlab and optionally Java. The internals of our submodularity high-performance framework were designed in an intuitive Object-Oriented (OO) fashion and we further covered it with a "zero-dependency" API that allows uncomplicated reuse from different platforms, e.g. Java. Furthermore, we provide a built-in and generic context-free Matlab adapter that allows invoking our high-performance SFM kernel implementations from Matlab and for unforeseen submodular function applications.

5. Evaluate the implementation on at least three different workload application scenarios and compare its performance against state-of-the-art implementations. We successfully delivered three application implementations: Minimum Graph Cut, Log Determinant and Text Corpus Selection in the context of Automatic Speech Recognition (ASR). Our final HPSFO outperformed in every case all other SFM implementations.

Figures 1.1, 1.2 and 1.3 show as example results the overall performance gains of our final HPSFO implementation compared to the Fujishige-Wolfe algorithm [2]. Our implementation is up to 32×, 30× and 509× times faster for the applications Minimum Cut, Log Determinant and Corpus Selection, respectively. The speed ups tend to increase with the problem sizes. Due to time limitation particularly while benchmarking Fujishige-Wolfe we could not compare for sizes beyond $n = 5000$.

1.2 Related Work

This thesis work is the continuation of a course project *Submodular function optimization for the s-t graph cut* [15] corresponding to the ETH course

[2]Note, however, that the base Fujishige-Wolfe implementation still benefits from indirectly using our high-performance foundation via the EO function evaluation.

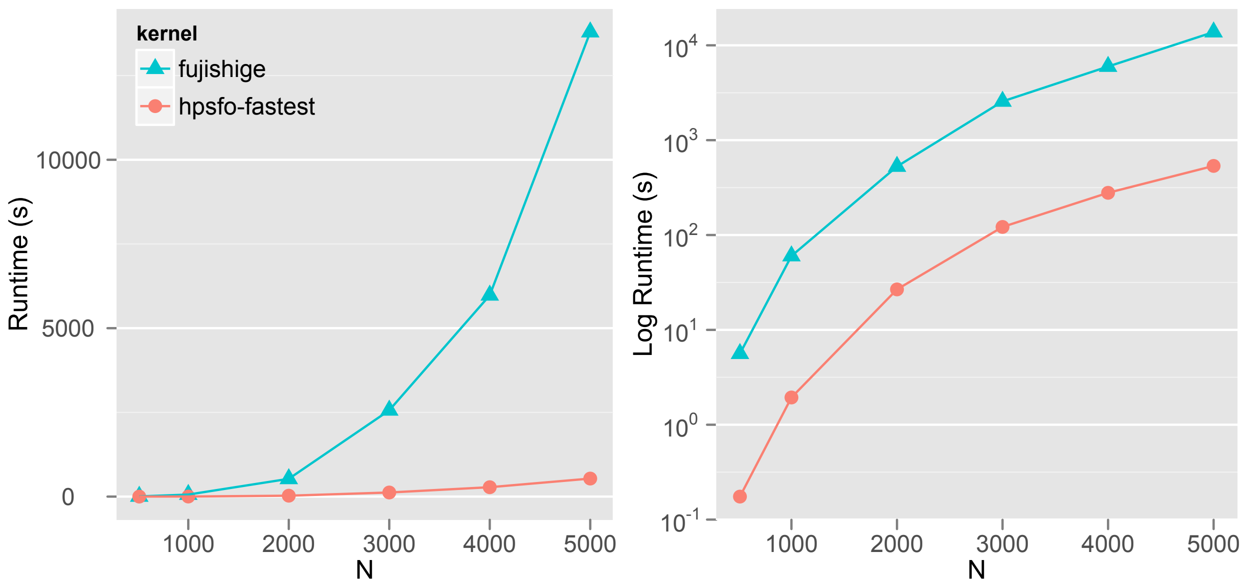


Fig. 1.1: Runtime(s) Minimum Cut Fujishige vs HPSFO-Fastest

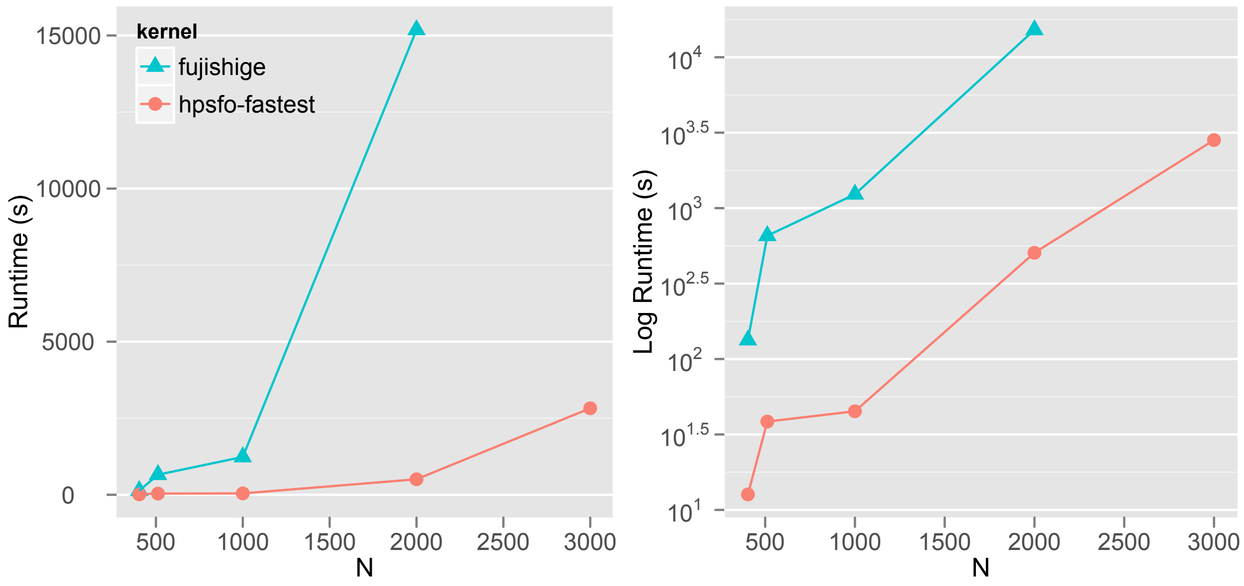


Fig. 1.2: Runtime(s) Log Determinant Fujishige vs HPSFO-Fastest

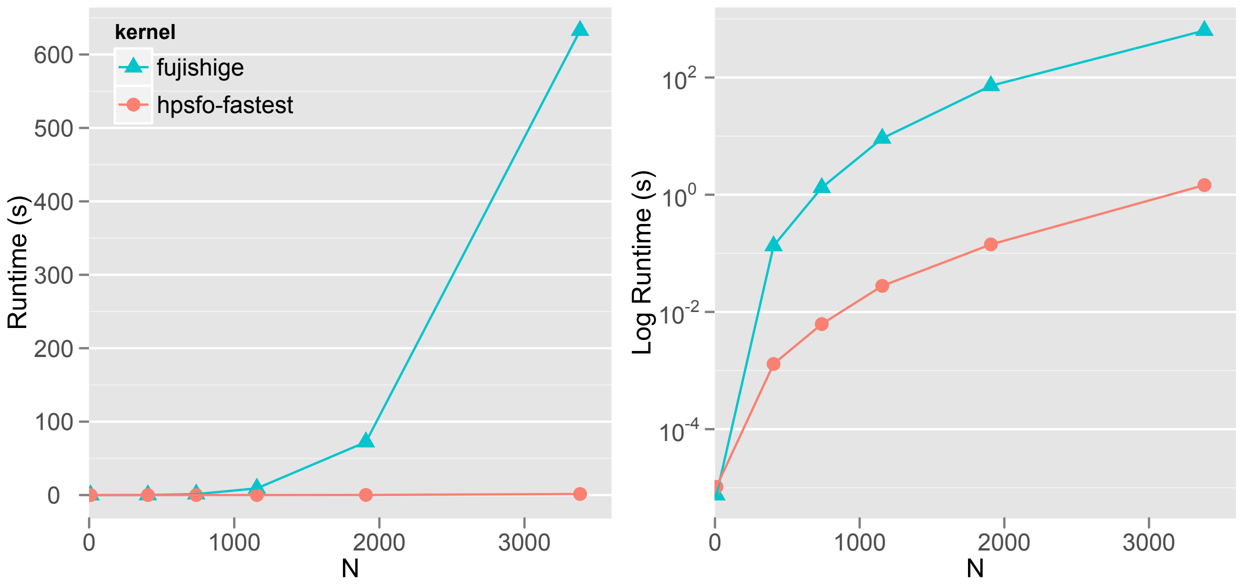


Fig. 1.3: Runtime(s) Corpus Selection Fujishige vs HPSFO-Fastest

How to Write Fast Numerical Code [40]. In that project we took as starting point the Matlab Toolbox for SFM from Prof. Krause [30], rewrote it in C and optimized it in multiple ways described in detail in the implementation chapter 4 and section 4.1.1. In this thesis work we continued where that project ended, and further optimized Krause MNP kernel as described in section 4.1.2 greatly reducing its complexity and cost by using fast orthogonal updates. Later, Satoru Fujishige kindly provided us with his MNP C kernel implementation [13] and after comparing how fewer floating point operations it would involve compared to Krause due to the simplicity of the original Wolfe algorithm, we turned the course of this work into optimizing Fujishige-Wolfe implementation too, resulting into our final and fastest HPSFO MNP kernel version. Both heavily optimized versions were built on top of a common high-performance infrastructure we implemented on top of Intel MKL.

In addition, we also reviewed Bach submodular Matlab Package [4] implementation and applied his approach (also present in Krause Toolbox) to establishing convergence via the duality gap associated with the convex Lovász extension [12] to the Fujishige-Wolfe and HPSFO implementations within our framework which would allow for convergence in the cases of Log Determinant and Text Corpus Selection applications. By studying Bach implementation we also extended our framework providing a configurable initial index permutation that would allow in a context-free fashion speeding up the optimization for some problems where a convenient choice of initial permutation (or simply random) would lead the search “closer” to the optimal and lower the number of iterations and thus, result in faster convergence.

Finally, we integrated Iwata scaling SFM algorithm into our submodularity framework but due to time constraints we could not get it to pass our test-suites for the three implemented applications beyond the basic Iwata test submodular function. It was nevertheless interesting to try this approach that while theoretically offers better complexity, practically it requires a “pre-scan” or a series of evaluations of the EO function proportional to the size of the ground set and in practice quickly get overtaken by our fast HPSFO implementation. Bach also in his MNP implementation does a similar series of evaluations “pre-scan” to compute a maximum norm required to build the ε used to test for duality gap convergence [4] we will discuss this further in chapter 2. In our HPSFO implementation we instead opted to make the ε configurable for the duality gap convergence criteria.

1.3 Thesis Structure

In chapter 2 we have provided an overview of the theory relevant for submodular function minimization and a self-contained description of the original Wolfe MNP algorithm including a detailed comparison, in particular, the algorithm algebra between Krause [30, 15] and Fujishige-Wolfe [13] MNP implementations. In that chapter we also present the modified Edmonds Greedy [11] that uses incremental update EO function evaluation.

The chapter 3 provides a brief problem statement for the three workload applications we implemented. Here we will also define the submodular objective functions and explain for each case the details of the incremental update EO evaluation concept.

We cover the implementation details related to performance optimizations and more in chapter 4. The chapter 5 provides Software and Object Oriented (OO) design details of our submodularity framework. Furthermore, the software framework chapter 5 serves as reference manual and extensibility guide including also the documentation for the generic Matlab adapter and integration concept.

Finally, the Experimental results chapter 6 provides a detailed performance comparison of the different SFM kernels and for all implemented applications.

Chapter 2

Submodular Function Minimization

In this chapter we introduce the mathematical underpinnings of our problem and describe the Minimum-Norm-Point (MNP) algorithm in detail. Furthermore, we will describe the supported MNP implementations and review their differences.

2.1 Overview of Submodularity

Let V be a *ground* set of cardinality n. A set function $f : 2^V \to \mathbb{R}$ is called *submodular* if it satisfies:

$$f(A) + f(B) \geq f(A \cap B) + f(A \cup B),\ \forall A, B \subseteq V. \tag{2.1}$$

An equivalent and more intuitive definition states that adding more elements to a set brings *diminishing returns*:

$$f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B), \forall A \subseteq B \subset V. \tag{2.2}$$

2.2 Submodular Function Minimization

Given the ground set V, we consider the core problem of finding an optimal subset $X \subseteq V$ that minimizes the submodular function $f : 2^V \to \mathbb{R}$:

$$\underset{X \subseteq V}{\text{minimize}}\ f(X) \tag{2.3}$$

We can cast many problems as a submodular function minimization (SFM) problem e.g., Clustering [31], MAP inference [31], Structured sparsity [31], etc. There are also many algorithms available for SFM e.g., M. Groetschel [18], S. Iwata [25], A. Schrijver [41] and J. B. Orlin [38]. In this work we will focus in optimizing the practically fastest algorithm for general SFM, the Fujishige-Wolfe MNP [13].

2.3 The Minimum-Norm-Point Algorithm

For the sake of completeness and self-containment we are going to include and describe here the details of the Wolfe MNP algorithm as well as some of the most important definitions and results from Fujishige work [13] and use it as reference throughout this work to review the most important algebraic differences between the two MNP implementations we chose to optimize: Krause [30] and Fujishige-Wolfe [13]. Furthermore, the algebraic details of the Fujishige-Wolfe algorithm included here will also help motivating the discussions on the different performance issues and improvements, and we will continuously refer to the specific Steps of this algorithm 1 in the remaining chapters.

The MNP algorithm was first introduced by Philip Wolfe [49] for finding the MNP in the convex hull of a given finite set of points in the n-dimensional Euclidean space R^n. Fujishige observed that this approach can be applied to the base polytope of submodular polyhedra, and as a consequence obtain a general approach to SFM [13] reusing the original Wolfe proposal to construct a practically fast SFM algorithm and it has been to date the practical algorithm of choice for general submodular function minimization.

For convenience, and in the context of our SFM problem let us first introduce the following definitions of affine hull and convex hull of a set C (Boyd [6]):

$$\textbf{aff } C = \{w^T x \mid x_1, \ldots, x_n \in C, e^T w = 1, e = (1, 1, \ldots, 1)^T\} \tag{2.4}$$

$$\textbf{conv} C = \{w^T x \mid x_1, \ldots, x_n \in C, e^T w = 1, e = (1, 1, \ldots, 1)^T, w \geq 0\} \tag{2.5}$$

Furthermore, let us also introduce the main definition and theorem from Fujishige work (Fujishige [12]):

Definition 1. (Fujishige [12]) Let E be a finite nonempty set and f be a submodular function on 2^E, i.e., $f : 2^E \to \mathbb{R}$ satisfies 2.1 for any $X, Y \subseteq E$.

We suppose that $f(\emptyset) = 0$ without loss of generality. We then define polyhedra

$$\mathrm{P}(f) = \{x \mid x \in \mathbb{R}^E, \forall X \in 2^E : x(X) \leq f(X)\} \tag{2.6}$$

$$\mathrm{B}(f) = \{x \mid x \in \mathrm{P}(f), x(E) = f(E)\} \tag{2.7}$$

$\mathrm{P}(f)$ is the *submodular polyhedron* and $\mathrm{B}(f)$ the *base polyhedron* associated with submodular function f on 2^E. Since $\mathrm{B}(f)$ is bounded, it is also a polytope.

Theorem 1. (Fujishige Theorem 3.2 [13] [12]) *Let x^* be the minimum-norm point in the base polyhedrom* $\mathrm{B}(f)$ *given by Def. 1. Define*

$$\mathrm{A}\_ = \{e|e \in E,\ x^*(e) < 0\}, \tag{2.8}$$

$$\mathrm{A}_+ = \{e|e \in E,\ x^*(e) \leq 0\} \tag{2.9}$$

Then, $\mathrm{A}\_$ *is the unique minimal minimizer of f, and* A_+ *is the unique maximal minimizer of f.*

Theorem 1 proves that if we find the MNP in the base polyhedron $\mathrm{B}(f)$, then we can "read off" a minimizer (in fact all minimizers) by taking the elements of the ground set corresponding to coordinates with non-positive entries. Because of this theorem we can directly use Wolfe MNP algorithm 1 to solve the submodular function minimization problem by means of the Greedy Edmonds algorithm [11]. The Wolfe MNP outlined in algorithm 1 maintains and updates a simplex S and at every step executes a linear optimization over the convex hull **conv** P of P 2.5. The Wolfe algorithm only works if this linear optimization can be efficiently done over the polytope $\hat{P}$. Wolfe's implementation in its original setting can not be efficiently applied to a general polytope Q because the number of extreme points of Q can be exponentially large with respect to the dimension n. A base polyhedron $\mathrm{B}(f)$ 1 associated with a submodular function f on 2^E is a class of polytopes where the linear optimization can be efficiently done even when the number of extreme points of Q is exponentially large with respect to dimension n. Edmond Greedy algorithm can be applied to a base polyhedra $\mathrm{B}(f)$ associated with submodular functions. (Fujishige-Isotani [13])

The Fujishige extended version of Wolfe algorithm 1 in the context of SFM takes as input a *ground set V* that contains all the set elements we wish to use to find an optimal subset from. The Step 1 of the algorithm 1 chooses any generated point p in P and adds it to the simplex S. Note that in the context of SFM the points set P that the Wolfe algorithm refers to is not represented explicitly but computed iteratively as shown in Step 2 of

Algorithm 1 Wolfe's MNP Algorithm (Fujishige-Isotani [13])

```
Input: A finite set $P$ of points $p_i$ $(i \in I)$ in $\mathbb{R}^n$
Output: MNP $x^*$ in the convex hull conv $P$ of the points $p_i$ $(i \in I)$.
 1: {Step 1} Choose any point $p$ in $P$ and put $S \leftarrow \{p\}$ and $\hat{x} \leftarrow p$.
 2: loop {Major}
 3:   {Step 2} Find any point $\hat{p}$ in $P$ that minimizes the linear function
      $\langle \hat{x}, p \rangle = \sum_{k=1}^{n} \hat{x}(k)p(k)$ in $p \in P$. Put $S \leftarrow S \cup \{\hat{p}\}$.
 4:   if $\langle \hat{x}, \hat{p} \rangle = \langle \hat{x}, \hat{x} \rangle$ then
 5:     return $x^* \leftarrow \hat{x}$
 6:   else
 7:     loop {Minor}
 8:       {Step 3} Find the MNP $y$ in the affine hull of points in $S$.
 9:       if $y$ lies in the relative interior of the convex hull conv $S$ then
10:         $\hat{x} \leftarrow y$.
11:         break {Minor}
12:       end if
13:       {Step 4} Let $z$ be the point that is the nearest to $y$ among the
          intersection of the convex hull conv $S$ and the line segment $[y, \hat{x}]$
          between $y$ and $\hat{x}$. Also let $S' \cup S$ be the unique subset of $S$ such
          that $z$ lies in the relative interior of the convex hull conv $S'$. Put
          $S \leftarrow S'$ and $\hat{x} \leftarrow z$
14:     end loop
15:   end if
16: end loop
```

Algorithm 2 Edmonds's Greedy Algorithm (Fujishige-Isotani [13][11])

```
Input: A weight vector $w \in \mathbb{R}^E$
Output: An optimal $x^* \in \mathrm{B}(f)$ that minimizes $\sum_{e \in E} w(e)x(e)$ in $x \in \mathrm{B}(f)$.

 1: {Step 1} Find a linear ordering $e_1, e_2, \ldots e_n$ of elements of $E$ such that
    $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_n)$.
 2: {Step 2} Compute $x^*(e_i) = f(e_1, e_2, \ldots, e_i) - f(e_1, e_2, \ldots, e_{i-1})$ $\{i =
    1, 2, \ldots, n\}$ .
```

the Greedy algorithm 2. In other words, the actual data the MNP algorithm operates on is generated dynamically by the evaluation of the oracle function (EO) using as input a given ordering of the elements of the ground set V. This is also an interesting performance issue that we can not run the MNP algorithm using warm data [40]. For that matter, pre-computing the function evaluation for all possible permutations of the ground set V is impractical and not needed, since the algorithm will not visit "uninteresting" surface areas of the implicit point set P. It is also very important to note that the choice of initial permutation of the ground set V elements has a strong effect in the algorithm i.e. an initial permutation of the ground set that generates a point p in P that is "closer to the optimal" will greatly reduce the number of iterations required to find the solution. The initial permutation of the ground set of choice in the implementations by Krause [30] and Fujishige [13] is the sequence $\{1, \ldots, n\}$ but it could be chosen randomly [4] or alternatively exploiting some property of the specific submodular problem at hand that could lead to a reduced number of iterations.

The Step 2 of the algorithm in a Newton-like descending fashion, forms a line segment towards the origin $[\hat{x}, (0, 0, \ldots, 0)]$ and picks a new point $\hat{p}$ which is the intersection between that line segment and the simplex S. If the hyperplane formed by $\langle \hat{x}, \hat{p} \rangle$ is a supporting hyperplane of the convex hull $\mathbf{conv}\, P$ then $\hat{p}$ is the minimum-norm point and the final solution. The test for that is shown at line 4 of Wolfe algorithm 1 or equivalently $\langle \hat{x}, \hat{p} \rangle - \langle \hat{x}, \hat{x} \rangle < \epsilon$ and this is one possible convergence criteria. (Fujishige-Isotani [13]).

The other convergence criteria we have studied and integrated as part of the Fujishige-Wolfe and our final HPSFO is the duality gap [12, 4]. The Lovász extension allows to reuse important results from convex analysis into submodular functions, in particular, the duality gap [12, 4]. The proposition 7.3 in Bach work [4] provides a convenient way to test for convergence in the MNP by means of the duality gap (Bach [4]). Assuming f is a submodular function, the condition to test is given by (Bach [4]):

$$\left(\min_{A \subset V} f(A) - \max_{s \in B(f)} s\_(V) \right) < \epsilon \tag{2.10}$$

and ε is computed in Bach implementation (Bach [4]):

$$\varepsilon = \sqrt{\frac{\sum_{i=1}^{n} (f(V) - f(V \setminus \{i\}) - f(\{i\}))^2}{n}} \cdot 1e^{-10} \tag{2.11}$$

we have therefore, reused this criteria to test for convergence in addition to the previous one. As we will discuss in further chapters, we have avoided the expensive computation shown in Eq. 2.11 but rather made the ε configurable. For practical purposes, we obtained the same results as Bach implementation by simply configuring $\varepsilon = 1e^{-10}$.

Definition 2. (Wolfe [49]) A point set P is *affinely independent* if no point of P belongs to the affine hull of the remaining points.

The Step 3 of the algorithm finds the MNP y in the affine hull of points spanned by S. Since S is affinely independent [49] see definition 2 finding the MNP in **aff** S is solved using the following quadratic system (Wolfe [49]):

$$\textbf{minimize } |x|^2 = w^T S^T S w \tag{2.12}$$

$$\textbf{s.t } e^T w = 1 \tag{2.13}$$

Forming the Lagrangian $w^T S^T S w + 2\lambda(e^T w - 1)$ and differentiating, the necessary conditions are obtained and they have a unique solution (Wolfe [49]):

$$e^T w = 1 \tag{2.14}$$

$$e\lambda + S^T S w = 0 \tag{2.15}$$

Once the solution to the system above w is found, we compute the MNP $y = Sw$. This algorithm step that involves finding the MNP y in the affine hull of points spanned by S by solving the Lagrangian set of equations depicted in Eq. 2.14 is the “Holy Grail” of the performance bottlenecks of the MNP algorithm and the main aspect around which most of the floating point operations (flops) end up being spent. This is also the reason why all along the execution of the algorithm we need to maintain and update a matrix R that mirrors S and offers a convenient structure to efficiently solve these systems. This step is also what makes the studied implementations different from each other. The details of how the different MNP implementations solve these equations is discussed in the next sections. Finally, once y is found and if y lies in the relative interior of the convex hull **conv** S, then we put $\hat{x} \leftarrow y$, break out of the minor loop and go back to Step 2 otherwise the algorithm moves onto Step 4. [13]

If the point y found in Step 3 falls outside convex hull **conv** S, then the Step 4 of the algorithm is executed. At this step the algorithm has made a “mistake” and it has to be corrected by removing one of the points p' previously added to the corral S. First, a new point z is found that is nearest to y among the intersection of the convex hull **conv** S and the line segment $[y, \hat{x}]$.

Then a subset $S' \subset S$ is found such that z lies in the relative interior of the **conv** S', in other words, a point p' that defines the simplex S is removed and is conveniently chosen in a way that z lies in the relative interior of **conv** S'. This step basically mutates the simplex matrix S and consequently its mirror R required for Step 3 as discussed before. This deletion of an arbitrary column point vector p' out of the matrices also critically affects performance and imposes some constraints in the design choices around representation and other aspects. This will be discussed in detail in chapter 4. (Fujishige-Isotani [13])

Definition 3. (Fujishige-Isotani [13]) A simplex S is called a *corral* if the minimum norm point in the affine hull **aff** S lies in the relative interior of the convex hull **conv** S.

The outermost loop at line 2 is known as the *major cycle* and the innermost loop at line 7 is known as the *minor cycle*. Every iteration of the major cycle increases the size of the simplex S by one, and every iteration of the minor cycle decreases the size of the simplex by at least one. The number of major cycles corresponds to the number of distinct points added to the simplex S. Similarly, the number of minor cycles can be viewed as the number of "mistakes" made by the algorithm: the number of points explored which do not end up being part of the end simplex S. The simplex S is a *corral* 3 whenever the algorithm goes from Step 3 to Step 2 as part of the major cycle. Every corral S uniquely determines the MNP $\hat{x}$ and every time the algorithm forms a new corral, the norm of the new $\hat{x}$ strictly decreases and this is the reason why the MNP algorithm has guaranteed convergence after a finite number of iterations. (Fujishige-Isotani [13])

Each time the Edmonds Greedy algorithm 2 is invoked, it executes n evaluations of the EO. It invokes the EO using as input a valid subset of the ground set V to be evaluated at once. An EO to be executed from the Edmonds Greedy can conveniently be made *stateless* e.g. the Iwata test function [13] [26] which would, in theory, enable parallel execution of the EO. In this work we propose a small but far reaching variation of Edmonds Greedy algorithm 3 based on the previous idea of Krause [30] where the EO is conveniently made *stateful* and the interface to invoke it changes to passing one set element at the time rather than a subset of elements, in a way that allows for *incremental update* of the submodular EO function. In other words, the function keeps a context or memory of the set of elements it has been invoked with so far and exploits the incremental structure of the underlying submodular function to *incrementally update* the evaluation using the new e_i rather than evaluating the function from scratch each time on the full subset. This seemingly trivial improvement offers massive performance improvements as we will see in the experimental results chapter 6 and better algorithmic complexity required in each evaluation of the EO e.g., effectively reduced the complexity of one of our applications EO from $O(n^2 \log\ n)$ to $O(n)$. Finally note how the new incremental evaluation variation algorithm requires the EO to *reset* or clear the context it has accumulated during one execution of the outer Greedy algorithm, this we can see in the Step 3 of the algorithm 3. This variation of the Edmonds Greedy algorithm requires only very minor unobtrusive changes to the Greedy step of the MNP algorithm. This approach resembles in a way to the differentiability concept in the continuous space, effectively offering a faster evaluation. The software design employed allows for flexibly introducing new function types that offer higher level of capabilities to be exploited by the MNP algorithm implementation.

Algorithm 3 Edmonds's Greedy with incremental update EO Algorithm [30]

```
Input: A weight vector $w \in \mathbb{R}^E$
Output: An optimal $x^* \in \mathrm{B}(f)$ that minimizes $\sum_{e \in E} w(e)x(e)$ in $x \in \mathrm{B}(f)$.

1: {Step 1} Find a linear ordering $e_1, e_2, \ldots e_n$ of elements of $E$ such that
   $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_n)$.
2: {Step 2} Compute $x^*(e_i) = f_{\text{inc}}(e_i) - f_{\text{inc}}(e_{i-1})$ $\{i = 1, 2, \ldots, n\}$.
3: {Step 3} Reset the function context or memory, invoke $f_{\text{reset}}()$
```

2.4 Implementations

The MNP algorithm implementations studied [13] [30] [4] have a very similar structure as described in the algorithm outline 1. There are only very few key differences that result in different performance and number of iterations required for convergence. We are going to cover the most important difference which is the way each algorithm solves the set of equations 2.14 as part of Step 3 of the Wolfe algorithm 1. Further details more related to the matrix representation etc will be described and discussed in detail in the implementation chapter 4.

2.4.1 Fujishige implementation

The method employed by Fujishige in his implementation [13] to solve the Lagrangian equations 2.14 is exactly the one described by Wolfe in [49]. We are going to include it here as well for completeness and use it as reference in following chapters to discuss the different performance improvements.

The matrix of the equations 2.14 is maintained using the upper triangular matrix R in the manner suggested by Golub and Saunders for the treatment of the least squares problem [16] where the orthogonal matrix Q of the QR decomposition is implicitly maintained, and e is the column vector $e = (1, 1, \ldots, 1)^T$ (Wolfe [49]):

$$ee^T + S^T S = (QR)^T QR = R^T \overbrace{(Q^T Q)}^{I} R = R^T R \tag{2.16}$$

Note that $R^T R$ is simply the Cholesky decomposition of $ee^T + S^T S$. Finally, the solution to the set of equations 2.14 becomes the result of solving the following two systems starting from Eq. 2.17. The MNP y of the affine hull **aff** S is obtained by multiplying the simplex S with the resulting w column vector (Wolfe [49]):

$$R^T \bar{w} = e \tag{2.17}$$

$$Rw = \bar{w} \tag{2.18}$$

$$y = Sw \tag{2.19}$$

Since R must be modified whenever S is, the steps of the Wolfe algorithm 1 are specialized as follows [1] (Wolfe [49]):

Step 1 The matrix R of dimensions 1×1 is initialized as $[(1 + |S_j|^2)^{1/2}]$

Step 2 Adding a new point p to the simplex S requires appending a column to R as well. This can be done by first solving the system below and appending the column $[r\ \rho]^T$ to R on the right:

$$R^T r = e + S^T S_j \tag{2.20}$$

$$\rho = (1 + S_j^T S_j - r^T r)^{1/2} \tag{2.21}$$

Step 4 Let j be the position of the component deleted, then delete the j^{th} column of R. Once a single column is deleted from R, it becomes an upper Hessenberg matrix [17] from the j^{th} column and R needs to be updated to be upper triangular again. The triangularization is implemented using a sequence of Givens rotations [17] and this explained in detail in the implementation chapter 4.

[1]Note that these steps simply correspond to updating and down-dating an existing Cholesky decomposition.

2.4.2 Krause implementation

The method employed by Krause in his original implementation [30] to solve the Lagrangian 2.14 is depicted in equations 2.22. The Eq. 2.22 computes the affine hull **aff** S by using a matrix space translation which reduces the dimension of S. The idea is to fix a point from the set in this case the column vector s_0 and compute the difference with respect to all other point column vectors, effectively creating the affine hull of S. The minimum-norm point y is computed by solving the system on the right operand of Eq. 2.23 and centering the result. Note that while the Eq. 2.23 is shown to invert the result of the matrix multiplication $\left(\hat{S}^T\hat{S}\right)^{-1}$, Krause implementation actually invokes the Matlab solve backslash operator and does not invert the matrix explicitly. Also note that in general the matrix $\hat{S}$ is not squared, this is why in the original Krause implementation it is multiplied by its transpose to get a square matrix and solve the system exactly. These two equations are much simplified in the actual high-performance version implementation which will discussed in the implementation chapter 4. In order to test whether the MNP y falls within the interior of **conv** S we employ Eq. 2.24 to get a representation μ of y in terms of S while enforcing that we have an affine combination i.e., $\sum_i \mu_i = 1$ which is the reason why we add the corresponding constraint to the system and we do so by appending a row of ones to S.

$$\hat{S} = \textbf{aff } S = \begin{bmatrix} s_{0,1} & s_{0,2} & \cdots & s_{0,k} \\ s_{1,1} & s_{1,2} & \cdots & s_{1,k} \\ \vdots & & \ddots & \vdots \\ s_{n,1} & s_{n,2} & \cdots & s_{n,k} \end{bmatrix} - \begin{bmatrix} s_{0,0} & s_{0,0} & \cdots & s_{0,0} \\ s_{1,0} & s_{1,0} & \cdots & s_{1,0} \\ \vdots & & \ddots & \vdots \\ s_{n,0} & s_{n,0} & \cdots & s_{n,0} \end{bmatrix} \tag{2.22}$$

$$y = s_0 - \hat{S}\left(\left(\hat{S}^T\hat{S}\right)^{-1}\hat{S}^T s_0\right) \tag{2.23}$$

$$\begin{bmatrix} s_{0,1} & s_{0,2} & \cdots & s_{0,k} \\ s_{1,1} & s_{1,2} & \cdots & s_{1,k} \\ \vdots & & \ddots & \vdots \\ s_{n,1} & s_{n,2} & \cdots & s_{n,k} \\ 1 & 1 & \cdots & 1 \end{bmatrix} \mu = \begin{pmatrix} y \\ 1 \end{pmatrix} \tag{2.24}$$

When y falls outside the convex hull **conv** S, this leads to Step 4 of the Wolfe algorithm 1 where a new point z is found that is nearest to y among the intersection of the convex hull **conv** S and the line segment $[y, \hat{x}]$. As part of this step we need to solve the following systems to get a representation λ of $\hat{x}$ in terms of S while again enforcing that we have an affine combination i.e., $\sum_i \lambda_i = 1$, here as well we enforce the new constraint by adding a new row of ones to S. Later λ is then used to find z in **conv** S that is closest to y.

$$\begin{bmatrix} s_{0,1} & s_{0,2} & \cdots & s_{0,k} \\ s_{1,1} & s_{1,2} & \cdots & s_{1,k} \\ \vdots & & \ddots & \vdots \\ s_{n,1} & s_{n,2} & \cdots & s_{n,k} \\ 1 & 1 & \cdots & 1 \end{bmatrix} \lambda = \begin{pmatrix} y \\ 1 \end{pmatrix} \tag{2.25}$$

To summarize, in Krause MNP kernel implementation we need to solve three systems depicted in equations 2.23, 2.24 and 2.25. The first system corresponding to Eq. 2.23 is determined and the other two systems corresponding to equations 2.24 and 2.25 are overdetermined. At this point it is readily apparent that Krause implementation involves a larger number of flops to execute Step 3 of Wolfe algorithm 1.

Chapter 3

Applications

We have chosen the following workload applications to test our high-performance implementation HPSFO kernel against all others. In this chapter we will briefly provide the problem statement and a few notes about the three applications. We will also explain the algorithm concept corresponding to the incremental update EO evaluation for each application. Note that these problems have already been extensively discussed in existing literature [4, 3, 31, 33, 28].

3.1 Minimum Graph Cut

The first application we employed to evaluate our implementation is the Minimum Graph Cut or *Minimum s-t Cut* problem, which is defined as follows. Taking as input a directed weighted graph $G = (V, E, w)$ with positive weights, and designated source and target nodes (s, t). It outputs two partitions $S \subset V$ and $T \subset V$ of vertices such that $s \in S$, $t \in T$ with $T = \{V \setminus S\}$. The goal is to minimize the sum of the weights of the edges that separate S from T. The Minimum Cut can thus be formulated as a SFM with the submodular function defined in Eq. 3.1 where $X^c = \{V \setminus X\}$ or the complement set of X.

$$f(X) = \sum_{u \in X,\ v \in X^c,\ (u,v) \in E} w_{u,v} - \sum_{(s,v) \in E} w_{s,v} \tag{3.1}$$

To ensure the constraint that s and t are in disjoint partitions, they are removed from the ground set and added as needed. The graph cut minimization with positive weights is a submodular problem [31] and we can employ the MNP algorithm to efficiently find a solution. While we use the minimum cut application to compare performance-wise different SFM implementations, it is important to note that there are much better algorithms for solving the minimum cut problem e.g. Edmonds Karp with general complexity $O(VE^2)$ [1] [43]. Since we use Boost and BGL as discussed in chapter 5 we have also provided a simple "zero-dependency" API to solve the minimum cut via the Edmonds-Karp algorithm.

The incremental update EO function version of the Minimum cut problem keeps the current set X and the result of the last evaluation of the EO function within its context. At every incremental step, it inserts into the current set a new element $X \leftarrow X \cup \{k\}$ and computes the new evaluation on top of the previous evaluation by adding the sum of the weights of the outgoing edges $(k, u) \in E,\ \forall\ u \in X^c$ and subtracts the sum of the weights of the incoming edges $(u, k) \in E,\ \forall\ u \in X$:

$$f(X \cup \{k\}) = f(X) + \sum_{(k,v)\in E,\ v\in X^c} w_{k,v} - \sum_{(u,k)\in E,\ u\in X} w_{u,k} \tag{3.2}$$

Therefore, the incremental update improves the EO evaluation complexity from $O(n^3)$ to $O(n)$ where we only need to make computations relative to the adjacency of the current node k. Note that in our implementation the non-incremental EO evaluation is $O(n^3)$ but it could be improved to $O(n^2 \log n)$.

3.2 Log Determinant

The second application we have implemented is the Log Determinant. The Log Determinant application has been described in detail in the work of Bach [4] but for a matter of self-containment we are going to summarize the problem description provided in Bach work [4] also here.

Given p random variables (RV) $X_1, \ldots, X_p$ that take finite number of values, we define $F(A)$ as the joint entropy of the variables $(X_{[k]})_{k\in A}$, this function is submodular. Its "symmetrization" version defined as $G(A) = F(A)+F(A^c)-F(B)$ is also submodular and leads to the mutual information between variables indexed by A and the those indexed by $A^c = \{V \setminus A\}$. This result can be applied to any distribution by means of differential entropies. In particular, we have the application for Gaussian RV, leading to the submodularity of the function defined through the Log Determinant $F(A) = \log |Q_{A,A}|$

[1] `http://www.boost.org/doc/libs/1_50_0/libs/graph/doc/edmonds_karp_max_flow.html`

for some positive definite matrix $Q \in \mathbb{R}^{p \times p}$. In the context of semi-supervised clustering, we are given p data points generated from a Gaussian process $x_1, \dots, x_p \sim \mathcal{N}(0; \Sigma)$ that are normally distributed with zero mean and covariance matrix Σ. Furthermore, we assume that any subset of points satisfy the same distribution assumption with covariance matrix $K_{[X,X]}$ where K is the $p \times p$ kernel matrix of the p data points, i.e., $K_{ij} = k(x_i, x_j)$ and k is the kernel function associated with the Gaussian process. Given a selected set A, we have A and A^c as two independent Gaussian processes with covariance matrices Σ_A and Σ_{A^c}, respectively. In order to maximize the likelihood under the joint Gaussian process, the best covariance matrix estimates are $\Sigma_A = K_A$ and $\Sigma_{A^c} = K_{A^c}$. This leads to a maximization problem that derives from the modular prior distribution on subsets $p(A) = \prod_{k \in A} \eta_k \prod_{k \notin A}(1 - \eta_k)$ and involves the negative log-likelihood of the two independent Gaussian processes and the mutual information between them. (Bach [4])

Finally, the semi-supervised clustering problem "two-moons" defined in Bach work [4] can be cast as a likelihood maximization problem that can, in turn, be represented as a SFM problem as shown Eq. 3.3 [4] where K is the kernel matrix, in particular, a Gaussian kernel $k(x, y) = \exp(-\alpha \parallel x - y \parallel_2^2)$ [4]. $K_{[A]}$ is the sub-matrix of K formed from the columns defined by the index set A.

$$f(A) = \underbrace{\log |K_{[A]}|}_{\text{Log-likelihood } A} + \underbrace{\log |K_{[A^c]}|}_{\text{Log-likelihood } A^c} - \underbrace{\log |K|}_{\text{Mutual information } I(A;A^c)} \tag{3.3}$$

The non-incremental EO function complexity and cost (in flops) corresponding to the Log Determinant application at every evaluation step is $O(n^3)$ where n is the dimension of the squared kernel matrix K, i.e., the complexity and cost is dominated by computation of the full Cholesky decomposition corresponding to $K_{[A]}$ and $K_{[A^c]}$ [17]. The incremental update EO evaluation version keeps a context containing the up-to-date Cholesky decompositions corresponding to $K_{[A]}$ and $K_{[A^c]}$. The context is initialized and reset to $A = \{\}$ and $A^c = \{1, \dots, n\}$, note that we compute the Cholesky decomposition of $A^c = \{1, \dots, n\}$ only *once* and store it as part of our context, the saved computation is copied into the appropriate matrix at every function reset call. At every incremental update EO function evaluation step a new element j corresponding to the column from K_j is processed, the Cholesky decomposition of $K_{[A]}$ is updated after appending the column K_j and conversely, the Cholesky decomposition of $K_{[A^c]}$ is down-dated after removing the column corresponding to K_j:

$$f(A \cup \{k\}) = \underbrace{\log|\mathrm{Cho}(K_{[A]})_{+k}|}_{\text{update Cho +col k}} + \underbrace{\log|\mathrm{Cho}(K_{[A^c]})_{-k}|}_{\text{update Cho -col k}} - \underbrace{\log|\mathrm{Cho}(K)|}_{\text{FV, computed once}} \tag{3.4}$$

Therefore, the incremental update improves the EO evaluation complexity and cost from $O(n^3)$ to $O(n^2)$ based on a series of Cholesky updates and down-dates [17].

3.3 Corpus Selection

Our third application is the Selection of Minimal Speech Corpora in the context of Automatic Speech Recognition (ASR) or as we call it Corpus Selection described in detail in the work of Hui Lin and Jeff Bilmes [33]. Again for completeness we provide here a minimal problem description.

The problem of corpus selection is that of finding the optimal subset from a set of utterances that simultaneously minimize the vocabulary size and maximizes the total amount of information, measured as either the cardinality of the utterances set, or a weighted scheme where the utterances are given a positive integer weight or alternatively, the duration of the speech. The problem can be modeled as a combinatorial optimization problem defined on a bipartite graph. Let V be the ground set of corpus of utterances, let F be the vocabulary set of distinct words contained collectively in these utterances. Then we can define the bipartite graph $G = (V, F, E)$ where $E \subseteq V \times F$ are the set of edges. Each $(v, f) = e \in E$ is an edge between an utterance $u \in V$ and a word $f \in F$ if utterance v contains word f. The goal is to find $X \subseteq V$ that maximizes the parametric (parameter λ) objective function (Hui Lin and Jeff Bilmes [33])

$$f(X) = w(X) - \lambda\Gamma(X) \tag{3.5}$$

or equivalently that minimizes the objective function which is a submodular function [33] and we can efficiently solve it using our HPSFO implementation:

$$f(X) = w(V \backslash X) + \lambda\Gamma(X) \tag{3.6}$$

The submodular objective function in Eq. 3.6 means we want to minimize the amount of information or duration of speech in the complement set of utterances $V\backslash X$ effectively maximizing the amount of information in utterances of set X and penalize the vocabulary size or the amount of distinct words included in these utterances. The penalization is parameterized with the trade-off coefficient λ which controls the vocabulary size of the optimal subset of utterances. The larger the λ value, the smaller the vocabulary size will be. (Hui Lin and Jeff Bilmes [33])

The non-incremental EO function complexity corresponding to the Corpus Selection application at every evaluation step is $O(n^2 \log n)$. The incremental update EO function keeps a context that includes the total amount of information contained in the current set X of utterances and an efficient set representation containing the distinct vocabulary associated with the current set of utterances. At every evaluation step the incremental EO simply updates the amount of information corresponding to the passed utterance k and inserts the associated vocabulary into the vocabulary set associated to X:

$$f(X \cup \{k\}) = w(V \backslash X) - \underbrace{w(\{k\})}_{\text{weight of } (s,k) \in E} + \lambda \underbrace{\Gamma(X \cup \{k\})}_{|\text{voc(X)} \cup \text{voc(\{k\})}|} \tag{3.7}$$

Therefore, the incremental update improves the EO evaluation complexity from $O(n^2 \log n)$ to $O(n)$ where the $O(n)$ originates from iterating the vocabulary (adjacent edges) associated to utterance k, and the cost of inserting new words into the vocabulary set is constant in our bitset implementation. Note that the n corresponding to the incremental update EO should intuitively be small, we are only looking at the adjacency, or words connected to utterance k but the Corpus Selection does not give hard bounds on the maximum number of words that one utterance can have. Therefore, it would be possible to be the worst-case, i.e., the graph having one or two utterances that are connected to all words and thus, e.g., have adjacency $n - 1$.

Chapter 4

Implementation

As previously mentioned in the introduction chapter 1 this thesis work is the continuation of a course project [15]. In that course project we took Krause's MNP kernel Matlab implementation as starting point and rewrote it in C. From there on, we applied many optimizations and branched out about ten different major release versions or baselines. All the optimizations made to Krause's kernel are relevant to this work and will be discussed in the section 4.1.

Later, Satoru Fujishige provided his MNP kernel implementation and we applied many optimization techniques that will be discussed in the corresponding section 4.2. We named the resulting implementation, the HPSFO MNP kernel.

4.1 Krause's MNP kernel optimization

4.1.1 Course project optimizations (Azua, Chothia, Frunza [15])

The following optimizations were covered as part of the course project *Submodular function optimization for the s-t graph cut* [15] corresponding to the ETH course *How to Write Fast Numerical Code* [40]. The ideas we explored and implemented in that project served as a base and foundation for this thesis work. Therefore, we have included the most important results here.

Initially, we rewrote Krause submodular Toolbox Matlab [30] implementation in C substituting Matlab functions with our own matrix and vector operations. This approach posed some challenges, e.g., what would be the best way to solve an overdetermined system which is done transparently in Matlab using the operator backslash?. Our first working implementation employed Gaussian elimination to solve the three systems of equations described in section 2.4.2 in every iteration of Step 3 and Step 4 of the algorithm 1. Solving a determined system corresponding to Eq. 2.23 was straightforward. However, in order to solve the overdetermined systems of equations 2.24 and 2.25 we used the standard Least Squares approach where we try to find the x that minimizes the error cost function:

$$\| Ax - b \|_2^2 \tag{4.1}$$

For the sake of self-containment we derive the optimal solution to the Least Squares problem below. Basically we want to find the x that minimizes error for the overdetermined system of equations, so we take the derivative with respect to x and set it to zero in order to find the saddle point:

$$\begin{aligned}
\frac{\partial}{\partial x} = 0 \quad &\rightarrow \quad \frac{\partial}{\partial x}\| Ax - b \|_2^2 = 0 \\
&\rightarrow \quad \frac{\partial}{\partial x}[(Ax-b)^T(Ax-b)] = 0 \\
&\rightarrow \quad \frac{\partial}{\partial x}(A^TAx^Tx - 2A^Txb - b^Tb) = 0 \\
&\rightarrow \quad 2A^TAx - 2A^Tb = 0 \\
&\rightarrow \quad A^TAx - A^Tb = 0 \\
&\rightarrow \quad \underbrace{\underbrace{A^TA}_{\text{MMM}}\,x = \underbrace{A^Tb}_{\text{MVM}}}_{\text{Gaussian solver}}
\end{aligned}$$

It is clear that this approach was quite expensive; in order to solve one system and apply Gaussian elimination, a MMM and a MVM are needed that cost $2nm^2 + 2nm$ (m - rows of A, n - columns of A). This solution lead to non-convergence for the largest test-cases and it was clearly due to errors propagating as the number of operations increased. The Gaussian elimination cost in flops is $\frac{2n^3}{3} + \frac{3n^2}{2} - \frac{7n}{6}$ [17].

We had several options to overcome the convergence issues described before. We chose QR decomposition as solution since for this particular case of solving an overdetermined system, the QR decomposition offers the best trade-off between cost and numerical stability compared to, e.g. SVD [39]. Another strong point towards choosing QR was the existence of well known blocked algorithms [8] designed for parallelizing the QR decomposition. We initially planned to employ these QR block algorithms in order to optimize for the memory hierarchies. We took as starting point the QR decomposition implementation in [39] and extended it to support non-squared matrices. The derived Eq. $A^TAx = A^Tb$ can be solved more cheaply via QR in the following way (substituting A by QR):

$$\begin{aligned}
A^TAx &= A^Tb \\
(QR)^TQRx &= (QR)^Tb \\
R^T\underbrace{Q^TQ}_{I}Rx &= R^TQ^Tb \\
\cancelto{I}{R^T}Rx &= \cancelto{I}{R^T}Q^Tb \\
Rx &= \underbrace{Q^Tb}_{MVM}
\end{aligned}$$

Finally, since R is upper triangular the final Eq. $Rx = Q^Tb$ can be solved using right substitution [39]. Note that in this case we bypass the costly MMM and MVM from the solution above. The QR decomposition is known to cost $2n^2(m - \frac{n}{3})$ flops [17] for the compact variation and $4(m^2n - mn^2 + \frac{n^3}{3})$ flops when computing Q explicitly [17]. In our initial implementation we computed Q^T explicitly.

At this point the algorithm converged and we were able to reproduce the same results (including number of iterations) as the Krause's Matlab MNP kernel code for all test-cases. Through profiling we found matrix operations to be consuming the majority of the runtime. The next optimizations focus on reducing this cost.

Our initial baselines employed a matrix representation of one dimensional contiguous memory in row-major ordering. This turned to be very inefficient due to the general behaviour of the MNP algorithm, namely for every iteration it continuously adds and removes points from the corral S, which in our implementation corresponds to physically appending and deleting column vectors from matrix S. We employed pre-processing instructions or macros and added support for both row-major and column-major representations toggleable at compilation time. As we progressed with further improvements we were able to test both each time and the column major representation always delivered superior performance.

We then introduced the Intel Math Kernel Library (MKL) [1] as means to

[1]`http://software.intel.com/en-us/articles/intel-mkl/`

solve the QR decompositions and focus our efforts on more specific bottlenecks of our problem. Our first implementation using MKL invoked the LAPACK `dgels` driver routine, which solves overdetermined or under-determined real linear systems using either QR or LQ factorizations. We took advantage of the fact that we solved two right-hand sides with the same matrix S corresponding to equations 2.24 and 2.25. By caching the computation of the QR done the first time and reusing it in the second we avoided doing duplicated work. We accomplished this by switching to the lower level API directly, using `dgeqrf` (QR factorization), `dormqr` (MVM with Q) and `dtrsm` (triangular solve). We only perform the $O(n^3)$ costly step (QR) once. We did not gain as much as we initially expected by reusing the QR factorization. We believe this is due to the function `dgels` being highly optimized for building the QR decomposition and solving the system in one step. Having resolve some of the major bottlenecks in matrix operations we shifted our focus to optimizing the function evaluation, which is specific to the graph-cut problem.

Initially, our graph was represented using the adjacency matrix format, but considering the sparse structure of a graph matrix and the lack of reuse during the computation we improved the representation of the graph to an adjacency list represented as a structure of arrays rather than an array of structures. This compact representation improved the locality of our EO function for the Cut problem with neighbouring nodes stored next to each other.

The EO functions receive a list of ground subset indexes and we often need to sort them so we can efficiently compute set differences which we do by invoking `std::set_difference` included in `algorithm` as part of the standard C++ library [29]. Initially and in the C version we implemented our own merge sort. However, we can exploit the fact that subset element identifiers are integers between 1 and n where n is the ground set size. The merge sort complexity is $O(n \log n)$ whereas in this specialized case of sorting integer elements we reduced it to $O(n)$. The pseudo-code for this integer *linear sorting* algorithm is depicted in algorithm 4.

Until now each function EO evaluation was done as prescribed by the Edmonds Greedy algorithm 2 as part of Step 2 of the MNP algorithm 1. The polyhedron Greedy results in a sequence of function evaluations with subsets ranging from the empty set to the full ground set V - for example:

$$f(\{\}), f(\{3\}), f(\{3,8\}), f(\{3,8,2\}), ...$$

Algorithm 4 Linear Sorting

```
Input: elems: Elements, num_elems: Size of elems
Output: $\text{elems}_i \leq \text{elems}_j \forall i \leq j$
 1: Find $\text{elems}_{\text{min}}, \text{elems}_{\text{max}}$
 2: Set last = $(\text{elems}_{\text{max}} - \text{elems}_{\text{min}})$
 3: Initialize temp[0 ...last] = 0
 4: for i=0; i < num_elems; i++ do
 5:   temp[$\text{elems}_i$ - $\text{elems}_{\text{min}}$] += 1
 6: end for
 7: for j=0, i=0; j ≤ last; j++ do
 8:   for k=0; k < temp[j]; k++, i++ do
 9:     elems[i] = $\text{elems}_{\text{min}}$ + j + k
10:   end for
11: end for
12: return elems
```

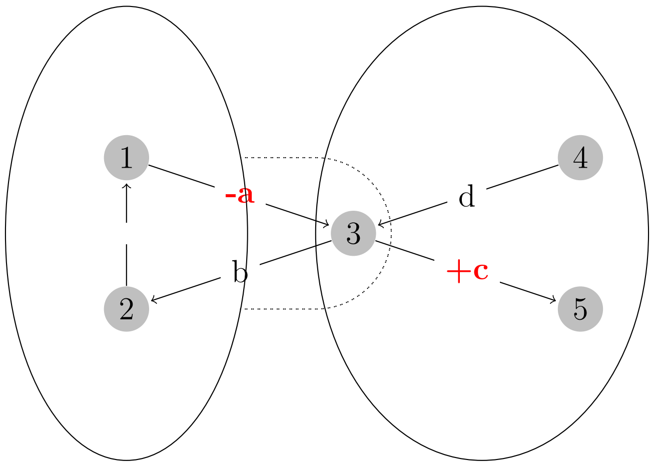


Fig. 4.1: Graph cut - Incremental evaluation example.

This approach is wasteful, e.g., when evaluating $f(\{3, 8, 2\})$ we inspect nodes 3 and 8, which were already considered in the previous evaluation of $f(\{3, 8\})$. We then implemented the improved strategy discussed before, applying Krause *incremental updates* of the EO evaluation depicted in algorithm 3 where given $f(A)$, we can compute $f(A \cup \{k\})$ by inspecting only node k's adjacency. The intuition behind this is illustrated figure 4.1. In order to include node 3 in the source partition, one needs to add the weight of edge c and subtract the weight of edge a from the previous cut value $(f(\{1, 2\}))$.

In contrast to the method suggested by J. Edmonds 2, this new solution using *incremental update* of the EO function corresponding to algorithm 3 only performs operations proportional to the in- and out-degree of a node. Further, since fewer edges are inspected we have fewer memory accesses, which are very expensive as there is no reuse to hide the effect of cache misses. The implementation of the incremental evaluation in this case requires bookkeeping the current set within a context. We employed a binary set representation, that offers constant time for setting an element as well as for testing existence of an element.

4.1.2 Further optimizations

As part of this thesis work further performance optimizations were applied to Krause MNP kernel implementation. First, all matrix and vector operations were rewritten in terms of Intel MKL [2] primitives and whenever possible take advantage of level BLAS-1, BLAS-2 or BLAS-3 operations. Software that relies on BLAS [3], LAPACK [4] or in this case Intel MKL is highly portable, and will typically run very efficiently. The higher the BLAS level, the more possibilities for exploiting locality, vectorization and parallelism. It is also common to trade higher flop count in exchange for higher BLAS level to the end of higher performance and better scalability [47]. Since this type of improvement is common to Fujishige too, it has been described in detail in section 4.3.1. Particularly relevant for Krause MNP kernel is the implementation of the subspace translated operation shown in Eq. 2.22 by means of `cblas_dger` [23]. We can efficiently write the subspace translation as a level BLAS-2 operation as shown in 4.2 and code listing 4.1 here we do an outer vector vector multiplication as depicted in Eq. 4.3 effectively creating a matrix of dimensions $n \times k$ as k replications of the column vector S_0. Multiplying the resulting matrix by $\alpha = -1$ and passing as input A the submatrix $\hat{S}(1, \ldots, k)$ we effectively converted the subspace translation or computation of the affine hull **aff** S to a BLAS-2 operation taking advantage of the optimizations for locality, vectorization and parallelism transparently offered by Intel MKL.

[2] `http://software.intel.com/en-us/articles/intel-mkl/`
[3] `http://www.netlib.org/blas/`
[4] `http://www.netlib.org/lapack/`

$$A = \alpha \cdot x \cdot y^T + A \quad \text{definition of } \texttt{cblas\_dger} \tag{4.2}$$

$$A = -1 \cdot \begin{pmatrix} s_{0,0} \\ s_{1,0} \\ \vdots \\ s_{n,0} \end{pmatrix} \cdot (1, 1, \dots, 1) + \begin{bmatrix} s_{0,1} & s_{0,2} & \cdots & s_{0,k} \\ s_{1,1} & s_{1,2} & \cdots & s_{1,k} \\ \vdots & & \ddots & \vdots \\ s_{n,1} & s_{n,2} & \cdots & s_{n,k} \end{bmatrix} \tag{4.3}$$

Listing 4.1: Subspace translated implemented in MKL BLAS-2

```
// one-time initialization
static tsfo_vector<T> ONES(tsfo_vector<T>::
   VECTOR_BUFFER_SIZE - 1, 1.0);

CBLAS_ORDER b_order = blas_order();
lapack_int m        = m_rows;
lapack_int n        = m_cols;
double alpha        = -1.0;
const double *x     = S.data();
lapack_int incx     = 1;
const double *y     = ONES.data();
lapack_int incy     = 1;
lapack_int lda      = m;
cblas_dger(b_order, m, n, alpha, x, incx, y, incy, a, lda);
```

Next, the matrix mutations, e.g., delete columns was optimized and this is also described in the appropriate section 4.3.3.

Another improvement to Krause MNP kernel was to simplify the step of finding of the MNP point y in the affine hull $\hat{S} = \textbf{aff } S$ as depicted in Eq. 2.23 corresponding to Step 3 of the algorithm. The simplification consists of avoiding the expensive MMM of $\hat{S}$ by its transpose $\hat{S}^T$ and the MVM corresponding to the multiplication of s_0 by $\hat{S}^T$, and instead solving the overdetermined system directly using the QR decomposition. The result was a major reduction in flop count and higher stability, the computation is done in the following way:

$$\hat{S}\hat{x} = s_0 \quad \text{multiplying both sides by } Q^T \text{s.t. } S = QR \tag{4.4}$$

$$Q^T \hat{S} x = Q^T s_0 \tag{4.5}$$

$$Rx = \underbrace{Q^T s_0}_{MVM} \tag{4.6}$$

then, the Eq. 2.23 simplifies to:

$$y = s_0 - \hat{S}\left(\left(\hat{S}^{-1}\right)s_0\right) \tag{4.7}$$

The last and greatest performance improvement to Krause MNP kernel was to avoid recomputing the full QR factorization of S and $\hat{S} = \textbf{aff}\ S$ at every Step 3. The goal ideally is to simply compute the QR factorization of S and $\hat{S}$ once (note that we need both $S = QR$ and $\hat{S} = \hat{Q}\hat{R}$) and update it as result of the matrix mutations due to the nature of the algorithm. Ideally we would like the full cubic-cost QR computation to happen only once across the whole execution of the MNP algorithm and from there on have only quadratic-cost updates, but we will see in a moment why this is not always possible for the case of $\hat{S} = \hat{Q}\hat{R}$.

We represent our matrices and vectors as contiguous 1-D memory chunks and the mutations required by the MNP algorithm do translate into actual memory movements, this will be explained in section 4.3. The matrix mutations consist of:

I **Append column**: corresponds to adding a new point column vector s_{k+1} to the point set matrix simplex $S_{n\times k}$ and this effectively happens at every Step 2 of the MNP algorithm 1.

II **Delete column**: corresponds to the case where the algorithm makes a "mistake" and needs to remove an existing point column vector s_j out of the point set $S_{n\times k}$ and this effectively happens at every Step 4 of the MNP algorithm 1.

III **Append row**: required by the specific way the MNP is implemented by Krause, we need to append a row of ones to the S matrix as shown in equations 2.24 and 2.25. We append a row of ones (or add a new constraint) to ensure that we get an affine combination, e.g., $\sum_i \mu_i = 1$. This happens at every Step 3 of Krause MNP implementation.

First we note that to append a column to S and therefore to $\hat{S}$, we do not require recomputing from scratch the subspace translated operation shown in 4.2. Every time a new column is appended to S we simply update the subspace translation in the way depicted in Eq. 4.8. Therefore, once QR is computed for S and $\hat{S}$, we can efficiently update the $S = QR$ and $\hat{S} = \hat{Q}\hat{R}$

corresponding to the append column matrix mutation. The details of the QR updates implementation are discussed in section 4.3.4.

$$\hat{S}_{n\times k+1} = \left[\begin{array}{cccc|c} \hat{s_{0,1}} & \hat{s_{0,2}} & \cdots & \hat{s_{0,k}} & s_{0,k+1} - s_{0,0} \\ \hat{s_{1,1}} & \hat{s_{1,2}} & \cdots & \hat{s_{1,k}} & s_{1,k+1} - s_{1,0} \\ \vdots & & \ddots & \vdots & \vdots \\ \hat{s_{n,1}} & \hat{s_{n,2}} & \cdots & \hat{s_{n,k}} & s_{n,k+1} - s_{n,0} \end{array}\right] \tag{4.8}$$

Next, the QR update corresponding to the delete column mutation can also be efficiently done and this is also discussed in section 4.3.4. However, here we note that in the specific case of deleting the point column vector s_0, destroys the existing **aff** $S = \hat{S}$ and requires recomputing the subspace translation of S from scratch and therefore recomputing its QR factorization $\hat{S} = \hat{Q}\hat{R}$. Every time the MNP algorithm finds it made a "mistake" Step 4 of Wolfe algorithm 1 and the point to delete is the column vector s_0 we have to recompute **aff** $S = \hat{S}$ and its QR $\hat{S} = \hat{Q}\hat{R}$.

Finally, appending one row to S in order to solve equations 2.24 and 2.25 can also be done more efficiently than recomputing the whole QR decomposition of S from scratch. In this case we do not even need to physically append the row to S which is very expensive due to the column-major matrix ordering of S. Instead we run the append row update directly to its QR, and use the updated QR to solve the equations, once the equations are solved we undo the append row QR update.

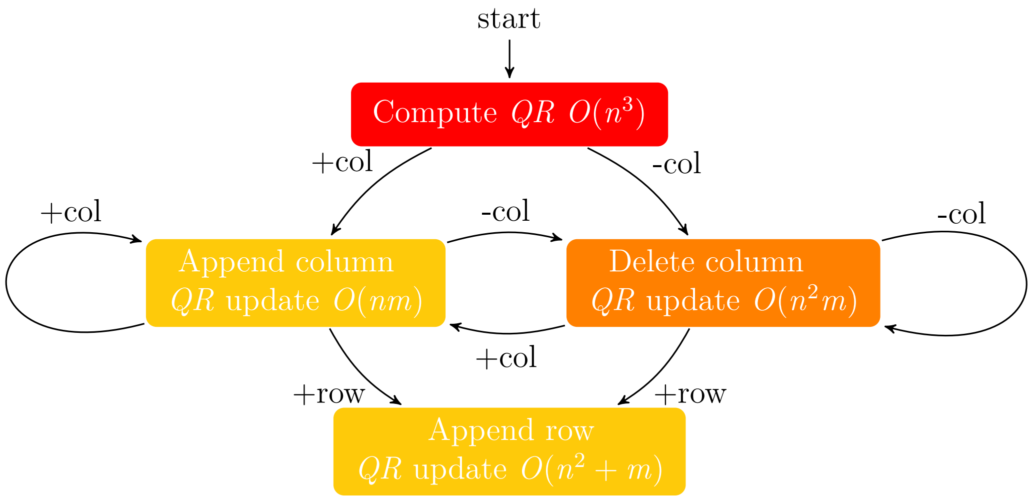

start
Compute QR $O(n^3)$
+col
-col
+col
-col
-col
Append column QR update $O(nm)$
Delete column QR update $O(n^2m)$
+col
+row
+row
Append row QR update $O(n^2+m)$

Fig. 4.2: Matrix mutations to $S_{m\times n}$ and corresponding QR updates

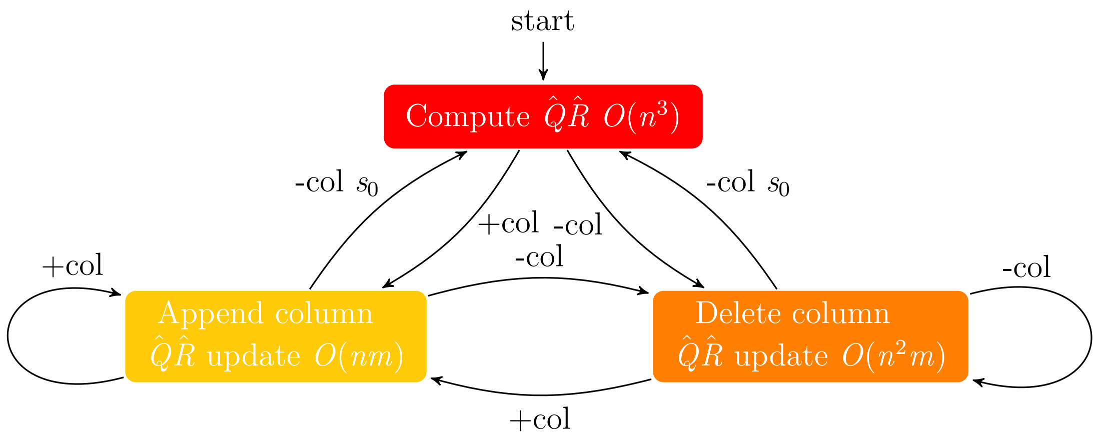

start
Compute $\hat{Q}\hat{R}$ $O(n^3)$
-col s_0
-col s_0
+col
-col
-col
+col
-col
Append column $\hat{Q}\hat{R}$ update $O(nm)$
Delete column $\hat{Q}\hat{R}$ update $O(n^2m)$
+col

Fig. 4.3: Matrix mutations to $\hat{S}_{m\times(n-1)}$ and corresponding $\hat{Q}\hat{R}$ updates

In Figures 4.2 and 4.3 we have simplified state diagrams of the possible matrix mutations to S and $\hat{S}$ and the resulting QR updates. The state color give an intuition of how much work and time is spent in that operation (the big O is also included). However, these are only simplified state diagrams because there is more to this updates "sandwiching", e.g., when we delete a column we need to build up and maintain an additional $Q_{\text{delete\_column}}$ that makes the R of $S_{n,-j}$ (that's it S with column j deleted) upper triangular again, and this $Q^T_{\text{delete\_column}}$ needs to be applied to any subsequent update, e.g., append column update or append row update. Similarly, whenever we append a new row we need to apply to this new row $Q^T_{\text{delete\_column}}$ and then build up an update $Q^T_{\text{append\_row}}$ that will make the R of $S_{n+1,k}$ upper triangular again. Finally, when we want to do a solve step we need to cumulatively apply all the updates to the right hand side column vector, e.g., $\hat{b} = Q^T_{\text{delete\_column}} Q^T_{\text{append\_row}} b$. In conclusion, separate from the flop count costs of the state changes depicted in Figures 4.2 and 4.3 we have also to take into consideration the overhead of updating and applying the orthogonal matrices that triangularize S or $\hat{S}$ after every update, namely bookkeeping the matrices $Q_{\text{delete\_column}}$ and $Q_{\text{append\_row}}$. Furthermore, the required memory footprint and memory copying resulting from bookkeeping these matrices adds up a non-negligible overhead to the overall solution. In conclusion, we can see this overhead reflected in Fig. 6.1 where we compare the original Matlab Krause MNP implementation to that implemented in C++ that includes all the optimizations described here and we can observe that the performance of the later degrades with bigger problem sizes, due to what we believe is one the one hand its lower Level BLAS of the updates, and on the other the high memory overhead connected to consistently maintaining all types of QR updates.

4.2 Fujishige's MNP kernel optimization

After we received a copy of Fujishige's MNP C kernel implementation and we realized algebraically how simpler it was implementing the Step 3 of the Wolfe algorithm 1 and the much lower flop count compared to Krause's implementation we decided to base our final and fastest MNP kernel HPSFO version using Fujishige's. We thoroughly tested his implementation and it didn't only outperform Krause's most optimized version but also demonstrated robustness by leading to convergence even in cases where the EO function had mistakes. It is also robust with respect to the sorting algorithm employed for implementing the Greedy algorithm step. Krause's implementation only works using `stable_sort` and Fujishige's implementation would work with any sorting algorithm, and this is one of the many improvements we did, using more efficient and even parallelizable sorting implementation.

We implemented the following improvements on top of Fujishige's C kernel and versioned it into our final HPSFO implementation.

For starters, we replaced all vectors and matrices with our high-performance template class type implementations `tsfo_vector<T>`, `tsfo_matrix<T>` and `tsfo_matrix_tria<T>` to take advantage of many aspects, e.g., contiguous and aligned memory to allow vectorization, pre-allocated and aligned buffer pooling, transparent parallelization, etc. By using abstraction the code became not only faster (as we will see later) but higher overall quality i.e., has much better communication of intent, it is a lot more readable, and thus easier to understand and maintain. The code listings 4.2 and 4.3 demonstrate the difference in the readability and maintainability of the code. Further, by abstracting the high-performance foundation using the class abstractions `tsfo_vector<T>`, `tsfo_matrix<T>` and `tsfo_matrix_tria<T>` that wrap Intel MKL, we will automatically gain free speed up due to future MKL continuous updates and improvements and the release of more advanced vector extensions, e.g., Advanced Vector Extensions 2 (AVX2) [5] [6].

Listing 4.2: Fujishige S and R matrices, computing MVM $x = Sw$

[5] `http://software.intel.com/en-us/avx/`

[6] `http://software.intel.com/en-us/articles/haswell-support-in-intel-mkl/`

```

// double linked-list of column indexes
ip = ivectoralloc(m_n + 2);
ib = ivectoralloc(m_n + 2);

// ...

// S and R matrices
ps = matrixalloc(m_n + 2, m_n + 2);
r  = matrixalloc(m_n + 2, m_n + 2);

// ...

/**************   step 1 ********************/
/****************** (A) ********************/

while (1) { /* 1000 */

  // MVM implementation navigating through the double-linked
  // list of column indexes structure ip
  i = ih;
  while (i != 0) {
    for (j = 1; j <= m_n; j++) {
      x[j - 1] += ps[i][j] * w[i];
    }
    i = ip[i];
  }
```

Listing 4.3: HPSFO S and R matrices, computing MVM $x = Sw$

```
// contiguous and aligned memory data, high-performance
// and high-abstraction!
tsfo_matrix     <double> S(n, 1, 0.0);
tsfo_matrix_tria<double> R(n, 1, 0.0);

// ...

/**************   step 1 ********************/
/****************** (A) ********************/

while (1) { /* 1000 */

  // tsfo_matrix::operator*() efficiently does an MVM
  // via Intel MKL's cblas_dgemv
  x = S*w;
```

Next, as shown in listings 4.2 and 4.3 replaced the main concept of the algorithm that deals with the matrix mutations from using a double-linked list of column indexes to instead maintain the matrix S and R physically as contiguous memory. In his implementation, Fujishige opted to avoid the expensive physical memory operations resulting from the MNP algorithm that requires adding and deleting points to and from the point set and corral S by means of a double-linked list of column indexes data structure. But as we will see in the chapter 6, the benefits of doing physical memory mutations by far outweighs its cost. We traded dealing with the expensive memory operations required to append and delete points with taking advantage of our high-performance foundation that builds on top of Intel MKL. We will discuss the way we leverage the cost of doing physical matrix mutations in section 4.3.3. Further, Fujishige implementation though more efficient mutating the main matrices S and R, suffered from very poor spatial and temporal locality, namely the column accesses in his implementation end up fetching arbitrary memory addresses leading to frequent cache evictions. Fujishige's code would not easily auto-vectorize and even if it did, the vectorization on unaligned memory is several times slower than that of aligned memory [10] [21].

Subsequently, we rewrote Fujishige's code casting his implementation in terms of Intel MKL (via our class abstractions `tsfo_vector<T>` and `tsfo_matrix <T>`). The main steps of 1 algorithm were identified and replaced with Intel MKL BLAS and LAPACK primitives, e.g., MVM was rewritten to use level-2 BLAS `cblas_dgemv` [7], replaced forward and backward solve steps to use level-2 BLAS `cblas_dtrsm` [8], etc. The code listings 4.4 and 4.5 show the result of applying this improvement to Fujishige's implementation. We note how we simplified and abstracted the two solve steps, we can also see a typical pattern of bundling computation, in this case computing the `usum` is bundled for the Fujishige implementation together with the back solve step, we on the other hand do this computation explicitly. More details on this will be discussed in section 4.3.

Listing 4.4: Fujishige find the MNP y in aff S as part of Step 3

```
// ...

/**************  step 3 ********************/

```

[7]`http://www.netlib.org/blas/dgemv.f`
[8]`http://www.netlib.org/blas/dtrsm.f`

```
while (1) { /* 2000 */
  // solve R^T\bar{u}=e
  i = ih;
  for (j = 1; j <= k; j++) {
    ubar[j] = 1.0;
    for (jj = 1; jj <= j - 1; jj++) {
      ubar[j] -= ubar[jj] * r[i][jj];
    }
    ubar[j] = ubar[j] / r[i][j];
    i = ip[i];
  }

  // solve Ru=\bar{u}
  usum = 0.0;
  i = it;
  for (j = k; j >= 1; j--) {
    u[i] = ubar[j];
    ii = ip[i];
    while (ii != 0) {
      u[i] = u[i] - r[ii][j] * u[ii];
      ii = ip[ii];
    }
    u[i] = u[i] / r[i][j];

    // bundle sum computation inside loop
    usum += u[i];
    i = ib[i];
  }

// ...
```

Listing 4.5: HPSFO find the MNP y in aff S as part of Step 3

```

  // ...

  /**************  step 3 *********************/

  while (1) { /* 2000 */
    // solve R^T\bar{u}=e
    tsfo_vector<double> ubar = R.solve_forward(ONES);

    // solve Ru=\bar{u}
    tsfo_vector<double> u = R.solve_backward(ubar, u);

    // compute sum with vectorization
    double usum = u.sum();

  // ...
```

We then, introduced support for OE with Incremental evaluation capability, so that the EO function evaluation as part of the Edmonds Greedy algorithm step becomes several times faster due to the lower complexity to evaluate it. The change is depicted in code listing 4.6

Listing 4.6: Incremental evaluation

```
// polyhedrom Greedy, generate a new extreme base
S.append_column(ZEROS);
sf_previous = 0.0;
min_F = DBL_MAX;

// use EO incremental capability f_inc(..) if available
if (sf_inc_context() != NULL) {
  sf_inc_context()->reset();
  for (int i = 0; i < n; i++) {
    sf_new = sf_inc_context()->f_inc(m_ground[s[i]]);
    S(s[i], k)  = sf_new - sf_previous;
    sf_previous = sf_new;
    min_F     = min(sf_new, min_F);
  }
} else {
  // otherwise simply use the standard EO f(..)
  assert(sf_context() != NULL);

  for (int i = 0; i < n; i++) {
    x.append(m_ground[s[i]]);
    sf_new = sf_context()->f(x);
    S(s[i], k)  = sf_new - sf_previous;
    sf_previous = sf_new;
    min_F     = min(sf_new, min_F);
  }
}
```

Further, we extended the convergence criteria of the separating hyperplane with the duality gap convergence previously discussed. Fujishige's implementation would otherwise not converge for some applications, e.g., the Log Determinant. However, the duality gap convergence also presents an expensive initialization step where exhaustive non-incremental evaluations proportionally large to the ground set size are needed, in order to compute a norm used to build the ε needed to test for the duality gap convergence.

We also replaced the Step 4 of Wolfe algorithm 1 that employs Givens rotations to repair the R matrix from upper Hessenberg to upper trapezoidal [17] after a point is deleted from the set. We replaced Fujishige's straightforward row Givens implementation with a handcrafted high-performance Register blocking and loop unrolling with lower flop count that in addition parallelizes with OpenMP (we also tested two other different alternatives), this point is discussed in detail in the section 4.3.4.

Additionally, we replaced Fujishige `quicksort2` implementation as part of the Greedy Step 2 of Wolfe 1 and reused the `parallel_sort` [9] implementation provided as part of Intel Threading Building Blocks [10] (TBB). But note that doing so leads to a different number of algorithm iterations. In order to compare the two generic kernels we kept Fujishige `quicksort2` and compared both implementations having HPSFO in single threaded mode, e.g., for the Gflops comparison. For the final "Fastest showcase" experiment section 6.5 we ran benchmarks of our final HPSFO implementation enabling parallelism and breaking compatibility with respect of the number of iterations and gaining further parallel speed up by using TBB `parallel_sort` implementation.

Finally, we kept the two versions, original Fujishige class type implementation [11] and the forked version HPSFO class implementation [12] into a reusable and extensible software design and framework that we will discuss in chapter 5. Note that for the sake of fair comparison , e.g., Gflops plots, we adapted the two improvements duality gap convergence and the incremental evaluation to the original Fujishige version `tfujishige_min_norm_point_kernel` so we can compare one-to-one the two generic SFM kernels i.e., to have exact same number of major and minor iterations and asymptotically the same complexity. It is important to point out that among all improvements, the incremental evaluation and its applicability to the different applications is the most important result of this work and can be seen as major complexity reduction on top of Fujishige work.

[9] `http://software.intel.com/sites/products/documentation/hpc/tbb/referencev2.pdf`

[10] `http://threadingbuildingblocks.org/`

[11] `tfujishige_min_norm_point_kernel`

[12] `thpsfo_min_norm_point_kernel`

4.3 High-performance foundation

4.3.1 Fast matrix and vector

The two bricks that lay down the high-performance foundations of our final MNP implementation are the `tsfo_vector<T>` and `tsfo_matrix<T>` implementations. We have successfully leveraged the complexity of the BLAS and LAPACK routines and more implemented by Intel MKL [13] by using these two template class abstractions. These two classes are however not meant to be generally reusable but rather evolved from a course project to the larger design it is now by fulfilling the different and specific use-cases of the MNP algorithm and, particularly, the need for super-fast computation and orthogonal updates. This we can see in some of the peculiar member function signatures which far from being general, help very specific use-cases like, e.g., the computation of the Log Determinant EO.

At the beginning of this work we evaluated the possibility to reuse an existing matrix framework that offered similar functionality to what we needed. The top choice at that time was the Open Source Eigen project [14]. The points that held us back from doing so are the following. Eigen is greatly optimized for evaluating complex expressions, and for this purpose they employ Expression Templates [2] [15], and the MNP does not have a strong use-case for complex expressions where the matrix and vector (vector in the sense of Matlab matrix and vector operations) evaluations would benefit from fusing into a single loop, how is statically done using Expression Templates, and this is one of the strongest areas of Eigen. In fact, to be able to statically optimize large expressions using templates they need to natively implement LAPACK operations like, e.g., `dgemm` rather than delegate such functions directly to the most optimized bleeding-edge performance Intel MKL implementation, here [16] we can see how the Eigen implementation is somewhat upper-bounded by Intel MKL for Level-2 or Level-3 BLAS operations that do not involve expressions, e.g., MVM or MMM; note that these benchmarks are for problems of small size and executed in a `Intel(R)Core(TM)2 Quad CPU Q9400 @ 2.66 GHz ( x86_64 )` processor architecture that does not support AVX. At the point of starting this implementation and for the very same reason stated before Eigen did not support AVX extensions. Further, Eigen did not offer functionality for orthogonal updates either. Therefore we decided to strive

[13] `http://software.intel.com/en-us/intel-mkl/`
[14] `http://eigen.tuxfamily.org/index.php?title=Main_Page`
[15] `http://eigen.tuxfamily.org/dox/TopicLazyEvaluation.html`
[16] `http://eigen.tuxfamily.org/index.php?title=Benchmark`

to our own vector and matrix implementations very tailored to the needs of the MNP algorithm and to take advantage of the fastest available LAPACK and BLAS routine implementations and more offered by Intel MKL.

The vector and matrix classes have as main attribute a contiguous 1-D memory location which they actually do not allocate or de-allocate as we will discuss later. These memory pointers are, however, declared in a compiler portable way to hint the compiler for AVX or SSE vectorization. We have implemented a generic means to declare aligned and restricted pointers as shown in listing 4.7. When we declare efficiently-aligned array or pointer types, the code that the compiler generates for these pointer arithmetic operations will often be more efficient than for other types [17] [21]. It is of course important that the way in which the memory is allocated complies to this declaration and we will see later how memory allocation is done framework-wide. An example of how we declare aligned pointer types, is in our `tsfo_vector<T>` class as shown in code listing 4.8. Note in this example that the T parameter is chained from the actual declaration of the vector type, e.g., `tsfo_vector<int >` in this case if we execute a canonical loop with potential for vectorization the compiler will, e.g., in an AVX platform execute 8 simultaneous integer operations per cycle using the appropriate vector registers [40].

Listing 4.7: Generic memory pointer declaration to hint vectorization

```
#if defined( __INTEL_COMPILER )
  template <typename T>
  struct sfo_type {
    typedef T* restrict __attribute__ ((align(32))) aptr32;
  };

#elif defined( __GNUG__ )
  template <typename T>
  struct sfo_type {
    typedef T* __restrict__ __attribute__ ((aligned(32)))
        aptr32;
  };
#endif
```

Listing 4.8: Example using the generic restrict aligned declaration

```
/**
```

[17]`http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Type-Attributes.html`

```
 * Concrete definition of vector that delegates most
 * operations to the Intel MKL LAPACK and BLAS routines.
 * Furthermore, it guarantees zero memory allocation
 * during execution outside static initialization.
 */
template<typename T>
class tsfo_vector {
private:
  typename sfo_type<T>::aptr32 m_data;
  int                          m_size;
```

In addition, the vector and matrix types offer a convenient and efficient abstraction and wrapper around many highly optimized Intel MKL functions, e.g., in listing 4.9 we see how the vector wraps fast vector Intel MKL functions. We have also provided convenience operators to increase readability of the code but only as long as the defined operators don't involve expensive memory copying that occurs when returning stack allocated objects by value [36], e.g., returning a matrix by value as it is the case of the MMM operation. The case of the MMM implementation is very interesting as we can see in code listing 4.10, here we would use it as `A.multiply(B, C);` intending to do `C = A*B + C` which is less readable than simply using the `operator*()` but we did so to avoid the otherwise poor performance that would result from returning C by value and therefore incurring into an expensive `operator=()` invocation. In C++11, this issue can be solved by either defining a move constructor or relying that the compiler would support and automatically do Named Return Value Optimization (NRVO). However, during development the top priority was not to have the most beautiful readable code but rather the most beautiful readable code that would not compromise performance in any way and by any chance and, e.g., the NRVO feature depends on the specific compiler. However, ensuring the move semantics would be a nice improvement that would increase the overall quality of the code.

Listing 4.9: Vector implementation of ln, min, max

```
/**
 * Computes the natural logarithm of elements of this vector
 */
template<>
inline tsfo_vector<double>& tsfo_vector<double>::ln() {
  if (size() > 0) {
    vdLn(m_size, data(), data());
  }

  return *this;
}

/**
 * Computes the max of a vector
 */
template<>
inline double tsfo_vector<double>::max() const {
  double result = 0.0;
  if (m_size > 0) {
    const lapack_int n    = m_size;
    const lapack_int incx = 1;
    int i = cblas_idamax(n, data(), incx);
    result = elem(i);
  }
  return result;
}

/**
 * Computes the min of a vector
 */
template<>
inline double tsfo_vector<double>::min() const {
  double result = 0.0;
  if (m_size > 0) {
    const lapack_int n    = m_size;
    const lapack_int incx = 1;
    int i = cblas_idamin(n, data(), incx);
    result = elem(i);
  }
  return result;
}
```

Listing 4.10: Matrix MMM multiply member function avoid operator*

```
/**
 * Performs a matrix-matrix multiplication of
 * (C += *this + B) and returns the result in the
 * output reference C.
 */
template<>
inline void tsfo_matrix<double>::multiply(const tsfo_matrix<
    double>& B, tsfo_matrix<double>& C) const {
  const tsfo_matrix<T>& A = *this;
  assert(A.cols() == B.rows());

  const int m = A.rows();
  const int n = B.cols();
  const int k = A.cols();

  C.rows(m);
  C.cols(n);
  if (m == 0 || n == 0 || k == 0)
    return;

  // C = alpha*A*B + beta*C
  const CBLAS_ORDER b_order = blas_order();
  const double alpha  = 1.0;
  const double beta   = 0.0;
  const MKL_INT lda   = A.leading_dim();
  const MKL_INT ldb   = k;
  const MKL_INT ldc   = A.leading_dim();

  cblas_dgemm(b_order, CblasNoTrans, CblasNoTrans, m, n, k,
      alpha, A.data(),
    lda, B.data(), ldb, beta, C.data(), ldc);
}
```

Another interesting point is how we solve and support the MNP algorithm use-cases and try to optimize the code with respect to the criteria of performance, readability, coding efficiency and abstraction. The example code in listing 4.11 corresponds to the Step 2 of the Wolfe algorithm 1 where we add a new point to the set and need to update the R matrix as shown in Eq. 2.20 Step 2 of points 2.4.1. The line 1 of this listing shows that we access a column element of matrix R via the `tsfo_matrix<T>::operator[]`, but when we acquire this reference, it doesn't involve a copy out of the matrix R_k column, instead the R memory address pointing to column k is wrapped within a `tsfo_vector<double>` (zero memory allocation remember!) and returned by reference [36]. In fact, `tsfo_matrix<T>` maintains a `std::dequeue` of `tsfo_vector<double>` that wrap column pointers for direct matrix column vector access and to cover use-cases like this one. Once the vector reference is acquired, it is used to forward-solve the system $R^T r_k = r_k$. We have carefully designed the matrix and vector abstractions to allow a column vector of a matrix to be passed directly as right hand-side vector to solve a system with that same matrix without having to create a copy of the column vector. The solution x must however, be copied to a separate column vector not to risk a race condition on the data while the underlying `cblas_dtrsm` invocation is done, e.g., with enabled parallelism and this is done in the `solve_forward` implementation. However, we see that the result copy x is assigned to R_k directly and the R_k column is updated with the result x with the implicit invocation of `tsfo_vector<T>::operator=`.

Listing 4.11:]HPSFO MNP implementation and matrix operator[]

```

// update R for the new S[k] by solving R^Tx=R[k] and
// saving it into R[k] i.e. R[k] = x
tsfo_vector<double>& rk = r[k];
rk = R.solve_forward(rk);
```

Our main matrix template implementation `tsfo_matrix<T>` transparently implements QR updates. The updates are applied if the method `tsfo_matrix <T>::qr_factorization()` is invoked at least once and then any of the matrix mutation methods `tsfo_matrix<T>::append_column`, `tsfo_matrix <T>::delete_column` or `tsfo_matrix<T>::append_row` are invoked. All the complexity of maintaining not only the main QR of the matrix but also the QR updates is hidden away from the client code, in this case the Krause algorithm implementation. The discussion on the structures and in general, the pseudo state machine required to consistently maintain multiple QR updates simultaneously is discussed in detail in section 4.3.4. We separated the fast triangularization for triangular matrices after deleting one column in the class implementation `tsfo_matrix_tria<T>` this way we have a convenient separation between the two competing methods for doing orthogonal transformations.

As we can see in previous listings we have used a template implementation not to lose generality, even though all Intel MKL routines require double precision type. When we define and use template types the implementations have to be available to all client code at compile time (rather than link time), which is accomplished by either including the `.h` file or `.cc .cpp` files , we strongly favored code *inlining* to minimize function calling overhead and at the same time create more opportunities for the compiler to optimize and rather “compromise” on the relatively larger sizes of the resulting binary library and applications.

4.3.2 Memory management

During the development as part of the course project, one of the improvements was to avoid memory allocation during algorithm execution by preallocating all the needed memory only once [36] and during static initialization. Note that Fujishige MNP implementation allocates all the needed memory within the algorithm implementation which is fine, but all this memory has to be allocated and de-allocated *each time* the Fujishige MNP kernel is invoked whereas in our solution the allocation is done *only once* during the whole execution of the library once is loaded, and allows reusing the same memory buffers across multiple executions of the HPSFO MNP algorithm. Our initial implementation of this idea resulted in a buffer pool that had key-labeled memory locations explicitly referred to by the algorithm at many code points. This solution gave us a major speed up but obviously didn't scale well with respect to supporting new algorithms or reusing the pre-allocated pool buffers in other areas, e.g., EO function evaluation. Therefore we created a reusable generic class type `tbufferpool<T>` that features pre-allocating and deallocating all the needed memory for computation i.e., double-precision matrix and vector memory and do so in an aligned way. Note that by using this simple memory pooling strategy other areas of the code don't become convoluted with the details of aligned memory allocation. Further, by instantiating the buffer pool to live in the global static scope, its destructor call is guaranteed by the static deinitialization [46] and so we don't need to worry about catching possible exceptions at all code end points and dealing with the resulting higher code complexity. Furthermore, matrix and vector instances are available without requiring any explicit static initialization to be invoked [18]. The declaration of class `tbufferpool<T>` is depicted in listing 4.12. It offers allocating a pool of memory optionally aligned in different ways, for example we align the vectors using AVX 32-byte address alignment and SSE 16-byte address alignment, further we align the memory used by our matrices to be memory address page aligned and minimize this way as much as possible chances of translation lookaside buffers (TLB) misses [9] [40] [7]. However, note that the memory addresses for vectors wrapping address pointers corresponding to matrix columns is in general not aligned, unless the number of rows is multiple of the alignment size.

Listing 4.12: Buffer pool class

```
enum talignment {
  PAGE_ALIGNED, SSE_ALIGNED, AVX_ALIGNED, NO_ALIGNMENT
};

template<class T>
class tbufferpool {
private:
  const long       m_initial;
  const long       m_size;
```

[18]Except the case in which the instantiation of the matrix or vector becomes part of an static initializer, but we do not have or support this use-case, in general the order in which static initialization initializes global variables is undefined.

```
  const talignment m_alignment;
  vector<T*>       m_queue;
  vector<T*>       m_all;

public:
  // constructor
  tbufferpool(long initial, long size, talignment alignment =
     NO_ALIGNMENT);

  // get next buffer element from the pool
  T* next();

  // release next element from the pool
  void release(T* buffer);

  void ensure_size(long size);

  // destructor
  virtual ~tbufferpool();
};

// constructor
template <typename T>
inline tbufferpool<T>::tbufferpool(long initial, long size,
   talignment alignment)
  : m_initial(initial), m_size(size), m_alignment(alignment) {
  assert(initial > 0);
  assert(size > 0);

  switch (m_alignment) {
    case PAGE_ALIGNED: {
      for (long i = 0; i < m_initial; ++i) {
        T* buffer = NULL;
        posix_memalign((void**) &buffer, sysconf(_SC_PAGESIZE)
           , m_size*sizeof(T));
        m_queue.push_back(buffer);
        m_all.push_back(buffer);
      }
      break;
    }
    case SSE_ALIGNED: {
      for (long i = 0; i < m_initial; ++i) {
        T* buffer = NULL;
        posix_memalign((void**) &buffer, 16, m_size*sizeof(T))
           ;
        m_queue.push_back(buffer);
        m_all.push_back(buffer);
      }
      break;
    }
    case AVX_ALIGNED: {
      for (long i = 0; i < m_initial; ++i) {
        T* buffer = NULL;
        posix_memalign((void**) &buffer, 32, m_size*sizeof(T))
            ;
        m_queue.push_back(buffer);
        m_all.push_back(buffer);
      }
      break;
    }
    case NO_ALIGNMENT:
    default: {
      for (long i = 0; i < m_initial; ++i) {
        T* buffer = new T[m_size]();
        m_queue.push_back(buffer);
        m_all.push_back(buffer);
      }
    }
  }
}
```

The code listing 4.13 demonstrates the use of the `tbufferpool<T>` class implementation. Here we see that the matrix does not explicitly allocate memory but instead wraps the pre-allocated memory address providing the abstraction to efficiently invoke Intel MKL primitives. Note as well that we used constants to initialize our buffer pool, but this would be inflexible since we would need to change some of these values namely the `MATRIX_BUFFER_SIZE` , and re-compile in order to run specific problem sizes. The solution to this drawback is to configure it via a pre-defined environment variable which can be read during static initialization with an appropriate function and API to let the submodularity library know what the maximum problem sizes are. We do so using the environment variable we defined as `SFO_MAX_M_N` which can be set using, e.g., Bourne Shell `export SFO_MAX_M_N=5000`. From the design standpoint this solution is a lot simpler because the different entities that require buffer pooling (matrix, matrix triangular, vector) do not need to be managed by a super-entity that knows and controls the memory allocation parameters across all of them. Each `tbufferpool` client can pull the maximum size directly from the environment and statically pre-allocate all the needed memory. We recognize this is not a perfect solution and different approaches may be employed as part of future work to make it more flexible and without compromising in performance. Note that any solution using a super-entity that pre-allocates all the memory in a static way, would need not only to compromise on the good Object Oriented (OO) design namely *encapsulation* [46] because this super-entity would need to become a `friend` of matrix and vector and have access to its private members but also it would require more complex code across all library end points, e.g., `benchmark`, `mincut` and in general all tests, e.g., `test_matrix`. Client applications of our library would also require to invoke this static initialization super-allocation and de-allocation explicitly. In conclusion, any alternative to the current design will compromise not only on the good OO design qualities but also significantly increase code complexity.

Listing 4.13: Matrix example using the bufferpool class

```
// static up-front allocation
template<typename T>
tbufferpool<T> tsfo_matrix<T>::s_bufferpool =
tbufferpool<T>(MATRIX_POOL_INITIAL, MATRIX_BUFFER_SIZE,
    PAGE_ALIGNED);

/**
 * reusable init method i.e., for all constructors
 */
template<typename T>
inline void tsfo_matrix<T>::init(tbufferpool<T>& bufferpool) {
  // ...

  m_data = bufferpool.next();

  // ...
}

/**
 * reusable release method i.e., subclasses
 */
template<typename T>
inline void tsfo_matrix<T>::release(tbufferpool<T>& bufferpool
    ) {
  // ...

  bufferpool.release(m_data);
  m_data = NULL;

  // ...
}
```

4.3.3 Fast matrix mutations

As discussed in the previous section 4.2 when we forked our HPSFO MNP kernel version the main and only drawback was the point column deletion from matrices S and R while we represent our matrices as contiguous 1-D memory because this involves physical and expensive memory movements using `memmove` [19] at every iteration of the algorithm. This use-case occurs as part of Step 4 of the Wolfe algorithm 1 when it finds it made a "mistake" that needs to be corrected by deleting an existing point from the corral S. We employed two different strategies to reduce the cost of these memory movements and the pay-off was very substantial in terms of performance.

The first improvements was to introduce a *Gaping* strategy. If we physically delete one column j from a matrix represented as contiguous 1-D memory in column-major ordering, our matrix gets "split" in two blocks: the first block from the beginning of the matrix until j and the second block from $j + \text{rows}$ until the end of the matrix $\text{cols} \times \text{rows}$. The physical deletion in this case would correspond to moving the second block from $j + \text{rows}$ to j and this we can do with `memmove`. However, if the column we delete is the first one, i.e., $j = 0$ we are effectively moving the entire matrix forward! and this leads to very poor performance. What we do then is to keep track not only of the memory address corresponding to the beginning of the matrix but also a gap shift that specifies how far ahead from the beginning address is the actual content of the matrix. Therefore, using this strategy and in the case of deleting $j = 0$ we don't need to move memory but only increase the memory address pointer. Similarly, deleting columns at the beginning of the matrix is cheap, since we only move small portions of memory. The final solution simply checks what half of the matrix the deleted column is in, if it is in the first half then the gaping is used and we move the first block forward, otherwise we move the second matrix block backward. For this strategy to work, we require $2\times$ the amount of memory we would normally do to store a matrix, since the content might shift forward and the end of the matrix could exceed the maximum size boundary. The idea is that if the gap shift overtakes the maximum matrix memory size we simply move the matrix content backward and tidy it back to have zero gap. The code listing 4.14 shows how we compute the address of a given element $a_{i,j}$ using the gaping strategy. One downside of Gaping worth mentioning is that the matrix memory address with a non-zero gap is no longer memory page-size aligned and, for that matter, no longer AVX or SSE aligned.

[19] `http://www.cplusplus.com/reference/clibrary/cstring/memmove/`

Listing 4.14: Matrix element access in column-major with gapping

```cpp
/**
 * Use the m_left_gap to keep track of how much the contents
 * of the matrix are shifted to the right due to physical
 * column deletions
 */
template<typename T>
inline T& tsfo_matrix<T>::elem(int i, int j) {
  return m_data[m_left_gap + j*m_rows + i];
}

template<typename T>
inline T& tsfo_matrix<T>::operator()(int i, int j) {
  return elem(i, j);
}
```

The second major improvement originates from the observation that when we physically delete a column from R i.e., an upper triangular matrix and in the case that the column j to be deleted is in the first half of the matrix so that the gaping strategy explained before applies; we do not require to move the first full matrix block forward, instead we only require moving the upper triangular elements and ignore the strictly lower triangular elements (which are, by the way, ignored in Intel MKL functions like, e.g., `cblas_dtrsm`). The major pay-off of copying over only the upper triangular memory originates from the fact that our matrices are always overdetermined, i.e., $m >> n$ so we have effectively reduced the amount of memory to move by a factor of m. The code listing 4.15 demonstrates the final implementation. At line 30 of this listing we can see how we efficiently copy the memory for the upper triangular case, in a spatial-locality friendly way, i.e., innermost loop to iterate over the rows of each column.

Listing 4.15: Matrix optimized delete column with gap and triangular

```cpp
 1 /**
 2  * Removes the column block specified by begin and end, very
 3  * efficiently moves data in column-major representation.
 4  */
 5 template<typename T>
 6 inline void tsfo_matrix<T>::delete_column_with_gap(int begin,
      int end, bool triangular) {
 7   assert(begin >= 0 && end >= 0);
 8   assert(begin <= end);
 9   assert(end < m_cols);
10
11   double ratio = (double) begin / (double) m_cols;
```

```
  assert(ratio <= 1.0);

  int block = end - begin + 1;
  assert(m_cols >= block);

  // deleted column is in second half of the matrix
  if (ratio > 0.5) {
    // move second block backwards
    if (end < (m_cols - 1)) {
      T* from = data() + m_rows * begin;
      memmove(from, from + m_rows * block, (m_cols - end) *
          m_rows * sizeof(T));
    }

  } else {
    // deleted column is in first half of the matrix
    int gap = block * m_rows;
    if (begin > 0) {
      // move first block forwards
      s_bufferpool.ensure_size(gap + m_left_gap + (m_cols -
          block)*m_rows);
      if (triangular) {
        for (int j = begin; j >= block; --j) {
          int jj = j - block;
          #pragma simd
          for (int i = 0; i < j; i++) {
            elem(i, j) = elem(i, jj);
          }
        }
      } else {
        memmove(data() + gap, data(), begin * m_rows * sizeof(
            T));
      }
    }
    m_left_gap += gap;

    // tidy back
    if (m_left_gap >= MAX_M_N*m_rows) {
      memcpy(data() - m_left_gap, data(), (m_cols - block)*
          m_rows * sizeof(T));
      m_left_gap = 0;
    }
  }
  m_cols -= block;
}
```

4.3.4 Fast orthogonal updates

As described in sections 4.1 and 4.2 we employ fast orthogonal transformations to either update (after matrix mutations) an existing QR factorization in the case of Krause MNP kernel implementation or re-triangularize the R of the implicit QR decomposition maintained in the Fujishige MNP kernel implementation.

In order to repair a broken upper triangular matrix we employ the two types of orthogonal transformations: Householder *reflectors* and Givens *rotators* [17] [48] and use them extensively to introduce zeros below the diagonal of a matrix effectively recovering an upper triangular matrix R which we need to efficiently solve a system. Orthogonal transformations are unitary i.e., they preserve the 2-norm and inner product after the transformation and algorithms built using them are norm-wise backward stable and therefore offer favorable error propagation properties [48]. Householder reflectors offer an excellent solution to zero out elements at "larger scale", e.g., all elements of a vector except the first, whereas Givens rotations is the method of choice to zero out elements more selectively.

In general, we have a column vector x as shown in example 4.9 and we would like to apply an orthogonal transformation H_i that will zero out all elements starting at element $i+1$.

$$H_i^T \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} \hat{x}_1 \\ \vdots \\ \hat{x}_i \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.9}$$

We can find such H_i using Householder reflectors. Let $v \in \mathbf{R}^n$ be nonzero, an n-by-n matrix P of the form $P = I - \frac{2vv^T}{v^Tv}$ is called a Householder reflector [17]. Taking v as shown in Eq. 4.10 we get a H_1 that will zero out or annihilate the elements $i \geq 2$ in example 4.9.

$$v = \begin{bmatrix} \|x\|_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} - \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \tag{4.10}$$

We can also find such H_i using Givens rotations, defined as G in Eq. 4.11

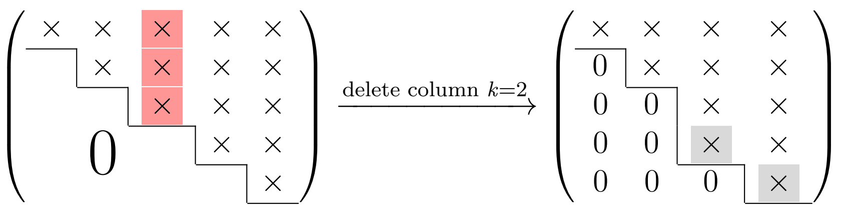
where $c = \frac{x_1}{\sqrt{x_1^2+x_2^2}}$ and $s = \frac{x_2}{\sqrt{x_1^2+x_2^2}}$ [17] and there are multiple ways to compute s and c, this way is the simplest but it doesn't guard against overflow. In general we can use Givens rotations to annihilate non-zero elements at any point in the matrix using the rank-2 changes of the Identity as depicted in Eq. 4.12. Finally, we can again build a H_1 that will annihilate the elements $i \geq 2$ in example 4.9 by rotating pairwise all elements from the bottom as depicted in Eq. 4.13 note that we annihilate from the bottom-up.

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \rightarrow G^T \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \hat{x}_1 \\ 0 \end{pmatrix} \tag{4.11}$$

$$G_{i,j} = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \tag{4.12}$$

$$\underbrace{G(1,2)^T G(2,3)^T \ldots G(m-1,m)^T}_{H_1} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} \hat{x}_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.13}$$

The Eq. 4.14 shows the effect of deleting an arbitrary column j from the matrix. The matrix becomes in upper Hessenberg form and needs to be triangularized by annihilating the two elements marked in light gray. The Eq. 4.15 shows the effect of appending one column at the end. In this case we need to annihilate the elements starting from $k = n + 1$ assuming we always have a determined or overdermined system. And last, in Eq. 4.16 we see the effect of appending one row (in our case a single row of ones as shown in equations 2.24 and 2.25) to an upper triangular matrix, it becomes the so-called 'triangular-pentagonal' matrix [32]. We will cover in the next sections 4.3.4 and 4.3.4 how these cases are handled in our implementation.



(4.14)

$$\begin{pmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & & \times & \times \\ & 0 & & \times \\ & & & \\ \end{pmatrix} \xrightarrow{\text{append column}} \begin{pmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & 0 & & \times & \times \\ & & & & \times \\ & & & & \times \end{pmatrix} \tag{4.15}$$

$$\begin{pmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & & \times & \times \\ & 0 & & \times \\ & & & \\ \end{pmatrix} \xrightarrow{\text{append row of ones}} \begin{pmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 1 & 1 & 1 & 1 \end{pmatrix} \tag{4.16}$$

QR factorization updates

In Krause's MNP kernel implementation, the QR decompositions for S and $\hat{S}$ are initially computed, and this cost accounts for the number of flops shown in Eq. 4.17 [17] and the actual cost of invoking Intel MKL `LAPACKE_dgeqrf` is shown in Eq. 4.18 [1].

$$C_{\text{flop}} = 2n^2(m - \frac{n}{3}) \tag{4.17}$$

$$C_{\text{flop}} = \left(2mn^2 - \frac{2n^3}{3} + 2mn + \frac{17n}{3}\right) \tag{4.18}$$

Subsequently if the point column vector s_0 is deleted, the full subspace translated $\hat{S}$ and therefore its $\hat{Q}\hat{R}$ will need to be recomputed. The QR decomposition is obtained by invoking the LAPACK routine provided in Intel MKL `LAPACKE_dgeqrf` and in LAPACK version 3.4.0 and later we have also the more interesting possibilities `LAPACKE_dgeqrt` and `LAPACKE_dgeqrt` that use Kressner's WY representation which offers better performance [32]. We didn't use the newer QR routines because Intel MKL released a version at the level of LAPACK 3.4.0 only after we had already implemented all the QR updates and the WY representation differs from the compact LAPACK QR representation we employed. The Eq. 4.19 provides an example of the compact QR LAPACK representation. It stores the R elements and Householder reflectors within a single QR matrix instead of two, it takes advantage from the fact that R is upper triangular and that the Householder reflectors can be conveniently compacted as elementary reflectors into the strictly lower triangular of QR. The trick however, is to leave the diagonal free for the $r_{i,i}$ elements by normalizing the elementary reflectors dividing by the first elementary coefficient of the reflector and storing it in a separate column vector named τ. Each Householder reflector can then be recovered by doing $P = I - \tau\frac{2vv^T}{v^Tv}$, but in practice we hardly need to this, since building all reflectors and accumulating them is expensive i.e., many MMM instead we use optimized LAPACK routines that will apply those compact elementary reflectors to a vector or a matrix, e.g., `dormqr`[20], `dlarf`[21], `dlarfx`[22]. It was important to understand well the compact QR representation in the context of the QR updates since we needed in several cases to pack updates this way , e.g., appending a new column.

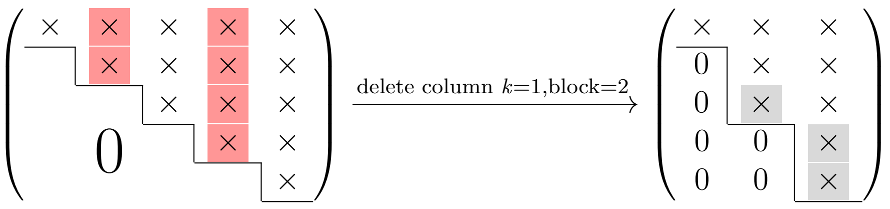
$$QR = \begin{pmatrix} r_{0,0} & r_{0,1} & r_{0,2} & r_{0,3} \\ v_{1,0}/v_{0,0} & r_{1,1} & r_{1,2} & r_{1,3} \\ v_{2,0}/v_{0,0} & v_{1,1}/v_{0,1} & r_{2,2} & r_{2,3} \\ v_{3,0}/v_{0,0} & v_{2,1}/v_{0,1} & v_{1,2}/v_{0,2} & r_{3,3} \end{pmatrix}, \tau = \begin{pmatrix} v_{0,0} \\ v_{0,1} \\ v_{0,2} \\ v_{0,3} \end{pmatrix} \tag{4.19}$$

At this point we are ready to discuss the different QR update algorithms tailored to the Krause MNP algorithm, e.g., taking into account the updates "sandwiching" and not in a more general setting. For a more comprehensive and general discussion on QR updates refer to the work of Sven Hammarling, Craig Lucas [19] and Daniel Kressner [23] [32].

The delete column QR update scenario shown in Eq. 4.14 is implemented by the the algorithm depicted in 5. Note that in this case we can also delete a block of p columns and not just one, e.g., Eq. 4.20. The cost in flops for this algorithm is depicted in Eq. 4.21 [19] where p is the number of columns deleted and k the index of the lowest column deleted, we can see that the column-block deletion update is lead by the cost of applying the old Q to the new matrix $S_{m\times(n-p)}$. The actual implementation of this algorithm was adapted and integrated from the existing Fortran implementation `delcols.f` and `delcolsq.f` [24] corresponding to that publication [19]. Note that the cost to apply r elementary reflectors in factored form $Q = Q_1Q_2\ldots Q_r$ to a matrix $C \in \mathbb{R}^{m\times k}$ is $2kr(2m-r)$ flops [17]. However, the actual cost in flops

[20]`http://www.netlib.org/lapack/double/dormqr.f`
[21]`http://www.netlib.org/lapack/double/dlarf.f`
[22]`http://www.netlib.org/lapack/double/dlarfx.f`
[23]`http://www.math.ethz.ch/~kressner/qrupdate.php`
[24]`http://www.maths.manchester.ac.uk/~clucas/updating/`

corresponding to `LAPACKE_dormqr` is $(4mnk - 2nk^2 + 3nk)$ [1].



(4.20)

$$C_{\text{flop}} = \underbrace{(4mn(n-p) - 2n(n-p)^2 + 3n(n-p))}_{\text{LAPACKE\_dormqr}} + 4np(\frac{n}{2} - p - k) + p^2(\frac{p}{2} + k) + pk^2 \tag{4.21}$$

Algorithm 5 Delete column QR update

Input: Begin index k, p number of deleted column(s) and existing QR decomposition of S (or $\hat{S}$).
Output: Updated $QR_{\text{delete\_column}}$

```
1: Delete existing $QR_{\text{delete\_column}}$ if present
2: Copy $QR_{\text{delete\_column}} \leftarrow S$ using cblas_dcopy
3: Apply existing $Q^T$ to $S$ in $QR_{\text{delete\_column}}$ using LAPACKE_dormqr.
   {Generate and apply $(n-k)$ householder reflectors}
4: for $j \leftarrow$ k, $n$ do
5:     Generate Householder reflector $H_j$ to annihilate non-zero elements below diagonal of column $j$ using dlarfg
6:     Apply the newly created $H_j$ to trailing columns $j+1 \ldots n$ using dlarfx
7: end for
```

The append column QR update scenario shown in Eq. 4.15 is implemented by the algorithm depicted in 6. The cost in flops for this algorithm is depicted in equations 4.22 and 4.23 corresponding to the append column update when there are no previous deletions and when there are previous column deletions respectively.

$$C_{\text{flop}} = \underbrace{(4mn + n)}_{\text{LAPACKE\_dormqr}} + \underbrace{2(m - n + 1)}_{\text{dlarfp}} \tag{4.22}$$

$$C_{\text{flop}} = \underbrace{(4mn+n)}_{\text{LAPACKE\_dormqr}} + \underbrace{2\times 2(m-n+1)}_{2\times\text{dlarfp}} + \underbrace{(n-k)\times\sum_{j=k}^{n} 4p(n-j)+(n-j)}_{(n-k)\times\text{dlarfx}} \tag{4.23}$$

$$= (4mn+n) + 4(m-n+1) + (n-k)(4p+1)\sum_{j=k}^{n}(n-j) \tag{4.24}$$

$$= (4mn+n) + 4(m-n+1) + \frac{(n-k)(4p+1)(k-n-1)(k-n)}{2} \tag{4.25}$$

In these equations we have that k is the minimum column index among all existing delete column updates and p is the total number of columns previously deleted. Furthermore, we assume that the cost in flops for `dlarfx` is $(4mn+n)$ [1].

It is readily apparent that the cost greatly depends on the specific workload i.e., if there are no column deletions or “mistakes”, then Krause implementation will perform very well. On the other hand, if we get accumulated deletion updates, it greatly increases the cost also for subsequent append column updates. Therefore, the gain in reduced flops compared to recomputing the QR decomposition from scratch each time will greatly depend on the specific workload application and how much the MNP algorithm makes “mistakes” or points that need to be deleted from the beginning of the matrix. One can see that the cost of this overhead together with lower Level-2 BLAS will not always win compared to a Level-3 BLAS QR decomposition.

The append row QR update scenario shown in Eq. 4.16 is implemented by the algorithm depicted in 7 that builds on top of Kressner’s `dbqru.f` routine which offers a more efficient tailored implementation of compact WY representation than a similar one built solely on top of LAPACK 3.2.0 routines [32]. The cost in flops for this algorithm is shown in Eq. 4.26 [32]. However, note that during this work, a new Intel MKL version was released including the newest LAPACK 3.4.0 that offers a new more efficient set of QR factorization and update routines using Kressner’s WY representation that solve the ’triangular-pentagonal’ QR update problem `DGEQRT` [25] and `DTPQRT`

[25] `http://www.netlib.org/lapack/explore-html/d2/dcf/dgeqrt_8f.html`

Algorithm 6 Append column QR update

```
Input: New column $S_{n+1}$ and existing $QR$ decomposition of $S$ (or $\hat{S}$).
Output: Updated $QR$ (and optionally $QR_{\text{delete\_column}}$) with new column
    $S_{n+1}$
 1: $R_{n+1} \leftarrow s_{n+1}$ using cblas_dcopy.
 2: Apply $Q^T$ to new column $R_{n+1} \leftarrow Q^T s_{n+1}$ using LAPACKE_dormqr.
 3: Generate Householder reflector to annihilate non zero elements below
    $(m-n)$ to $R_{n+1}$ using dlarfp
 4: if There is a valid $QR_{\text{delete\_column}}$ update then
 5:   Copy new column $R_{n+1}^{\text{delete\_column}} \leftarrow R_{n+1}$ using cblas_dcopy.
      {Apply $Q_{\text{delete\_column}}^T$ to the new updated column}
 6:   Set $r \leftarrow R_{n+1}^{\text{delete\_column}}$
 7:   for $j \leftarrow$ begin, $n$ do
 8:     Apply individual reflector $H_j$ encoded in $QR_{\text{delete\_column}}$ to column
        vector beginning at $r_j$ using dlarfx.
 9:   end for
10:   Generate Householder reflector to annihilate non zero elements below
      $(m-n)$ to $R_{n+1}^{\text{delete\_column}}$ using dlarfp
11: end if
```

[26]. Moving to this newer implementations was no longer possible due to time constraints, note that moving to these LAPACK 3.4 [27] routines involves a larger change in the actual QR representation to be WY and affects the other two algorithms 6 and 5.

$$C_{\text{flop}} = n^2 \overset{1}{\cancel{p}} + 4m\overset{1}{\cancel{p}} \tag{4.26}$$

In conclusion, for the Krause MNP kernel we extensively used the Householder reflectors solution. On the one hand because it would annihilate many non-zero elements at once, e.g., append column case with $m >> n$. On the other hand, it is easier and more maintainable to employ one single approach to store, copy and apply the transformations, i.e., the compact LAPACK QR format discussed before rather than mixing QR formats with that of an explicitly accumulated set of Givens rotations $Q = G_1^T G_2^T \dots G_m^T I$.

[26]`http://www.netlib.org/lapack/explore-html/d1/d55/dtpqrt_8f.html`

[27]`http://www.netlib.org/lapack/lapack-3.4.0.html`

Algorithm 7 Append row QR update

```
Input: New row, existing $QR$ (or $QR_{\text{delete\_column}}$) decomposition of $S$ (or $\hat{S}$).
Output: Updated $QR_{\text{append\_row}}$ corresponding to the new row.
1: if There is a valid $QR_{\text{delete\_column}}$ update then
2:     Copy $QR_{\text{append\_row}} \leftarrow QR_{\text{delete\_column}}$ plus an extra row using
       cblas_dcopy.
3: else
4:     Copy $QR_{\text{append\_row}} \leftarrow QR$ plus an extra row using cblas_dcopy.
5: end if
6: Invoke dbqru.f [28] that generates and applies a sequence of $n$ Householder
   reflectors and stores the result into $QR_{\text{append\_row}}$ in compact format.
```

Triangular matrix updates

As part of Step 4 of the Fujishige-Wolfe MNP algorithm 1 and when a "mistake" is made, we need to delete one point column from S, e.g. Eq. 4.14. This update requires re-triangularizing the R matrix after one column deletion see Step 4 in 2.4.1. In this case Wolfe algorithm employed Givens rotations to annihilate the non-zero elements below the diagonal. After the fast triangularization update was implemented we were very happy we could reuse the exact same fast Givens delete column update procedure for the implementation of the Log Determinant application with incremental evaluation where we down-date the existing Cholesky decomposition of set $A^c = \{V \setminus A\}$. In fact, due to the behavior of the MNP algorithm in the Log Determinant application where most of the time is spent updating and down-dating the Cholesky decomposition of the incremental EO is where our handcrafted *Register blocking with loop-unrolling and parallel* Givens implementation excels.

As discussed before, and in order to re-triangularize a matrix after a delete column update using Givens rotations we rotate two matrix elements, e.g. Eq. 4.11, then we need to propagate the transformation to the trailing columns. The self-contained example code in listing 4.16 demonstrates the concept and the listing 4.17 does the same but using Intel MKL and the LAPACK primitives `cblas_drotg` [29] that generates the sin and cosine that rotates the elements a and b; and the routine `dlasr` [30] that applies the rotation to a vector. In fact, there are multiple ways to generate the sins and cosines and we conveniently use them in the appropriate scenario:

[29] http://www.netlib.org/blas/drotg.f
[30] http://www.netlib.org/lapack/patch-3.0/src/dlasr.f

1. `drotg` fastest way to generate a Givens rotation
2. `dlartg`[31] generate Givens rotation with higher precision
3. `dlartgp`[32] generate Givens rotation with guaranteed positiveness of the upper element or in other words, guarantees positiveness of the diagonal elements of a matrix.

Listing 4.16: Basic Givens triangularization example

```
// create a simple matrix in column major order
const int ROWS = 3;
const int COLS = 3;
double matrix[ROWS][COLS];

// set 1's in the upper triangular and 2's in the band
// below the diagonal, now we want to annihilate the 2's
// using Givens rotations
matrix[0][0] = 1; matrix[1][0] = 1; matrix[2][0] = 1;
matrix[0][1] = 2; matrix[1][1] = 1; matrix[2][1] = 1;
matrix[0][2] = 0; matrix[1][2] = 2; matrix[2][2] = 1;

for (int j = 0; j < COLS; j++) {
  double a = matrix[j][j];
  double b = matrix[j][j + 1];

  // generate rotation P that annihilates element (j + 1, j)
  double h = sqrt(a*a + b*b);
  double c = a / h;
  double s = b / h;

  // apply the rotation to trailing columns
  for (int jj = j; jj < COLS; ++jj) {
    a = matrix[jj][j];
    b = matrix[jj][j + 1];

    double x = c*a + s*b;
    double y = c*b - s*a;

    matrix[jj][j]     = x;
    matrix[jj][j + 1] = y;
  }
}
```

[31] `http://www.netlib.org/lapack/explore-html/dd/d24/dlartg_8f.html`
[32] `http://www.netlib.org/lapack/explore-html/df/dc2/dlartgp_8f.html`

Listing 4.17: Basic Givens triangularization example using Intel MKL

```
// create a simple matrix in column major order
const int ROWS = 3;
const int COLS = 3;
double matrix[ROWS][COLS];

// set 1's in the upper triangular and 2's in the band
// below the diagonal, now we want to annihilate the 2's
// using Givens rotations
matrix[0][0] = 1; matrix[1][0] = 1; matrix[2][0] = 1;
matrix[0][1] = 2; matrix[1][1] = 1; matrix[2][1] = 1;
matrix[0][2] = 0; matrix[1][2] = 2; matrix[2][2] = 1;

double cc[COLS];
double ss[COLS];

for (int j = 0; j < COLS; j++) {
  double a = matrix[j][j];
  double b = matrix[j][j + 1];
  double c = 0.0;
  double s = 0.0;
  double r = 0.0;

  // generate rotation P that annihilates element (j + 1, j)
  cblas_drotg(&a, &b, &c, &s);

  // replicate the sins and cosines
  for (int jj = j; jj < COLS; ++jj) {
    cc[jj] = c;
    ss[jj] = s;
  }

  // apply the rotation to trailing columns
  char side       = 'L'; // Left, A := P*A
  char pivot      = 'V'; // Variable pivot
  char direct     = 'F'; // Forward
  lapack_int m    = ROWS - j;
  lapack_int n    = COLS - j;
  lapack_int lda = ROWS;
  dlasr(&side, &pivot, &direct, &m, &n, &cc[j], &ss[j], ((
      double*) matrix) + j*ROWS + j, &lda);
}
```

Taking a closer look at the two example listings 4.16 and 4.17 we should note that the matrix is in column-major order and to apply the update to the trailing columns we iterate over the columns at stride of size m, this we see in lines 23 and 39 of the listings 4.16 and 4.17 respectively. In other words, the memory access pattern has very poor spatial locality and reuse [9] [7] since the memory accesses hop in stride-m elements each time, and unless the problem sizes are small enough to fit in cache entirely, the cache will be evicted at every step due to capacity (or conflict) misses[33] [7] having a cache miss rate of one, i.e., one miss for every memory access. As soon as we get matrices of higher dimension this becomes a real problem and we will be spending most of the time wasting cycles due to cache misses and the cost of transferring memory to cache. A possibility to overcome this issue would be to simply switch to row-major ordering but this is not really an option for the MNP algorithm 1 since we require point column elements to be contiguous in memory so we can efficiently remove and append them whether physically or via pointers. We nevertheless, tried using row-major ordering and overall the performance was much poorer.

The big question then became, can we do better than this? Seeking an answer to this question we implemented three different approaches using the same main idea of Register blocking with loop-unrolling, there has been interesting research in this area [37] [47]. In a nutshell, we fix a block of size N_B and place it logically on top of the diagonal of the matrix, example in Fig. 4.27 shows how this stencil is done for two blocking steps with $N_B = 4$, here we load all the non-zero elements of the red block into registers, we then generate and apply Givens rotations on registers for that red diagonal block and save it back to the matrix. Note that there is an anti-diagonal dependency for the elements of this red block as we can see in Fig. 4.4 where the arrow $i \leftarrow j$ means that element j requires element i to be computed first. By solving the diagonal red block first that has anti-diagonal dependencies, we can do the more interesting sweeps to the green row block. This way we have effectively reused multiple generated rotations sins and cosines in registers for the diagonal red block and we can reuse and apply them at once to the trailing columns row green block in a anti-diagonal dependency-free fashion, and thus significantly increase performance. Fig. 4.4 shows the concept as "diagonal conquer", we try to conquer the diagonal block elements first, then apply the rotations deeper along the elements of each trailing column in a spatial locality friendly way. In Eq. 4.27 we highlight every two vertical elements used for generating the rotations, the gray one is the element we want to annihilate by rotating it with the red element above. Now we have

[33]Assuming a set-associative cache, e.g., 8-way set associative.

N_B rotations we can apply *at once* to every column of the row block depicted in green. Here is the important detail and trade-off; the larger the block size N_B, the deeper down the rows of each trailing column we can apply the Givens rotations at every step and the more we can reuse by filling the cache lines since this is precisely the contiguous memory direction of column-major ordering, but at the same time the more we run into the issue of register-spilling [37] while computing the red diagonal block. Note that we need $\left(\frac{N_B(N_B+3)}{2} + 2N_B\right)$ registers for every choice of N_B, we were able to gain speed up until $N_B = 8$. We implemented three variations of this main idea that are shown in points 4.3.4. In conclusion, by introducing blocking, we have improved the spatial locality and reduced the cache miss rate to, e.g., $\frac{1}{8}$. However, using Givens rotations we access $N_B + 1$ column elements only once and then move on, so there is not really a lot of reuse.

$$\begin{pmatrix}
\times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & 0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & 0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & 0 & 0 & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & 0 & 0 & 0 & \times & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & 0 & 0 & 0 & 0 & \times & \times & \times & \times & \times & \times & \times & \times \\
0 & 0 & 0 & 0 & 0 & 0 & \times & \times & \times & \times & \times & \times & \times \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \times & \times & \times & \times & \times & \times \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times & \times & \times & \times & \times \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times & \times & \times & \times \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times & \times & \times \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times & \times \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix} \tag{4.27}$$

Choice #1 Register blocking and OpenMP Load the diagonal block into registers, generate and apply Givens rotations to the whole diagonal red block in registers and update the matrix. Now iterate over the trailing columns and update $N_B + 1$ rows of each column at once. Furthermore, applying the Givens rotation to the trailing columns can be done in parallel and we used OpenMP for this, gaining up to $2\times$ parallel speed up for some applications, e.g., the Log Determinant application.

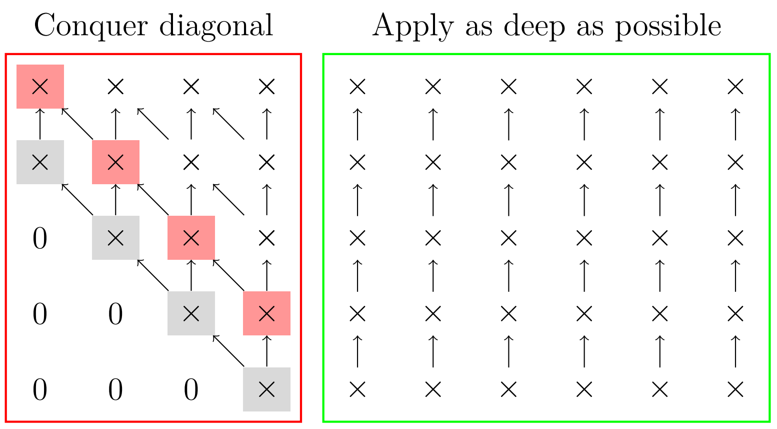


Fig. 4.4: Conquer diagonal block to apply deeper to trailing columns $N_B = 4$

Choice #2 Register blocking and accumulated rotations level 3 BLAS Load the diagonal block into registers and conquer it, i.e, generate and apply Givens rotations to the diagonal red block in registers and update the matrix. Now accumulate all the individual rotations in registers into an orthogonal matrix i.e. $G_{N_B}^T \dots G_2^T G_1^T I = H$. Create a patch of the row green block and apply H to the row block, saving the result of the computation directly on the main matrix via MMM i.e., Intel MKL `cblas_dgemm` [34]. The MMM API forces us to do $C = \alpha \cdot AB + \beta \cdot C$ which means that C is our matrix, A is our H and B has to be a patch containing the block row, i.e., we can't have B to be our main matrix. Therefore this requires a large memory operation, i.e., extracting the row block patch. A discussion has been started proposing to have an additional simpler Intel MKL `dgemm` API to support in-place MMM $A = \alpha \cdot AB$. [35]

Choice #3 Register blocking and transposition auto-vectorization Load the diagonal block into registers, generate and apply Givens rotations to the whole diagonal red block in registers and update the matrix. Now create a patch out of the matrix transposing the row block. Apply the rotations to this block using auto-vectorization and save the block back to the matrix. This requires two large memory operations i.e. extracting and saving back the row block patch.

[34] `http://www.netlib.org/blas/dgemm.f`
[35] `http://software.intel.com/en-us/forums/topic/277788`

The code listing 4.18 shows the Registers blocking with loop unrolling for solving the red blocks depicted in Eq. 4.27, this code is common to the 3 Choices discussed in points 4.3.4. We applied this same stencil up to $N_B = 16$ but we were able to gain speed up only until $N_B = 8$. There are a few interesting points to this listing. First, note that the specific rotation generator is not hard-coded into the function. The `GENROT` is a template parameter that allows subclassing or generating a new method/class and most importantly *inlining* each specific rotation primitive, as we have previously discussed there are several choices for generating rotations and we need the flexibility of switching between them without the performance hit of doing a function call each time. Second, we do not do *peeling* of the blocking, instead we let the algorithm overflow by at most $N_B - 1$, the reason is simply to avoid the increased code complexity of handling the peeling border cases and its bad locality (verified using Intel VTune [36]). Note that here we do not incur in illegal memory accesses, we have during initialization generously allocated enough memory to fit this overflow according to the maximum problem sizes we are going to run. However, we do incur into uninitialized value accesses, which leads to two possible problems: signalling NaN and accessing denormalized floating point values [10]. Doing computation on denormalized floating point values can incur in a performance hit [10], therefore as shown in line 24 of listing 4.18 we initialize the exact overflowed accesses to zero. However, in our experiments we could verify that without initializing this memory we would have the highest performance. Finally, note that the blocking offers improved locality, good temporal locality since the diagonal block is loaded once and the registers reused multiple times; and also good spatial locality because the N_B accessed columns in this block will be conveniently loaded into the cache-lines.

Listing 4.18: Register blocking Givens with loop-unrolling

```
1 template<typename T> template<typename GENROT>
2 inline void tsfo_matrix_tria<T>::triangularize4(int begin, int
      block, GENROT genrot) {
3 tsfo_matrix_tria<double>& r = *this;
4
5 const int m = r.rows();
6 const int n = r.cols();
7
8 double x00, x01, x02, x03, x10, x11, x12, x13, x21, x22, x23,
     x32, x33, x43;
9 double xx0, xx1, xx2, xx3, xx4;
```

[36] `http://software.intel.com/en-us/intel-vtune-amplifier-xe`

```
double u00, u01, u02, u03, u10, u11, u12, u13, u21, u22, u23,
    u32, u33, u43;
double uu0, uu1, uu2, uu3, uu4;
double c0, c1, c2, c3, s0, s1, s2, s3, d;
int im1, ip1, ip2, ip3;
int jp1, jp2, jp3;

int nb = 4;
assert(nb == TRIANGULARIZE_NB);
int n_iter = (n - begin)
       ? (n - begin) / nb
       : (n - begin) / nb + 1;
int nb_end = begin + n_iter*nb;

// avoid uninitialized value accesses by zeroing out
memset(r.data() + m*n, 0, m*(nb_end - n + 1)*sizeof(T));

int i = begin - block + 2;
int j = begin;

// blocked step
for (int k = 0; k < n_iter; ++k, i += nb, j += nb) {
 im1 = i - 1;
 ip1 = i + 1;
 ip2 = i + 2;
 ip3 = i + 3;

 jp1 = j + 1;
 jp2 = j + 2;
 jp3 = j + 3;

 // make the stencil as easy as possible
 x00=r(im1,j); x01=r(im1,jp1); x02=r(im1,jp2); x03=r(im1,jp3);
 x10=r(i,  j); x11=r(i,  jp1); x12=r(i,  jp2); x13=r(i,  jp3);
               x21=r(ip1,jp1); x22=r(ip1,jp2); x23=r(ip1,jp3);
                               x32=r(ip2,jp2); x33=r(ip2,jp3);
                                               x43=r(ip3,jp3);

 // make nb steps ahead using registers only
 genrot(&x00, &x10, &c0, &s0, &d);
 u00 = c0*x00 + s0*x10;
 u10 = c0*x10 - s0*x00;

 u01 = c0*x01 + s0*x11;
 u11 = c0*x11 - s0*x01;

 u02 = c0*x02 + s0*x12;
 u12 = c0*x12 - s0*x02;

```

```
u03 = c0*x03 + s0*x13;
u13 = c0*x13 - s0*x03;

x11 = u11;
x12 = u12;
x13 = u13;
genrot(&x11, &x21, &c1, &s1, &d);
u11 = c1*x11 + s1*x21;
u21 = c1*x21 - s1*x11;

u12 = c1*x12 + s1*x22;
u22 = c1*x22 - s1*x12;

u13 = c1*x13 + s1*x23;
u23 = c1*x23 - s1*x13;

x22 = u22;
x23 = u23;
genrot(&x22, &x32, &c2, &s2, &d);
u22 = c2*x22 + s2*x32;
u32 = c2*x32 - s2*x22;

u23 = c2*x23 + s2*x33;
u33 = c2*x33 - s2*x23;

x33 = u33;
genrot(&x33, &x43, &c3, &s3, &d);
u33 = c3*x33 + s3*x43;
u43 = c3*x43 - s3*x33;

// save back to matrix
r(im1,j)=u00; r(im1,jp1)=u01; r(im1,jp2)=u02; r(im1,jp3)=u03;
r(i,  j)=u10; r(i,  jp1)=u11; r(i  ,jp2)=u12; r(i  ,jp3)=u13;
              r(ip1,jp1)=u21; r(ip1,jp2)=u22; r(ip1,jp3)=u23;
                              r(ip2,jp2)=u32; r(ip2,jp3)=u33;
                                              r(ip3,jp3)=u43;
// ...
```

The Choice #1 Register blocking and OpenMP in points 4.3.4 after the Register blocking (red) step where the diagonal dependencies are solved, simply iterates along the trailing columns corresponding to the green row block, applying the sines and cosines already in registers to the $N_B + 1$ row elements of every column. The sample code corresponding to $N_B = 4$ is shown in listing 4.19. First, note that we load $N_B + 1$ rows for each column into registers, apply the rotations and save it back. Second, note that the loop iterates hopping in a stride-m pattern and therefore it can not take advantage of auto-vectorization or for that matter intrinsics, we need contiguous memory access for that. Note as well that touching the top element `r(i, jj);` might or might not lead to a cold cache miss [40] due to the structure of the stencil, as we can see in Eq. 4.27 the top row of the current sweep overlaps with the row of the previous sweep. However, for all the block sizes (we tried up to $N_B = 16$), the row elements of the column jj will definitely fit into cache and depending on the matrix size we will get at most one cold cache miss at every iteration. Finally, note that the OpenMP configuration is set to `runtime` to be read from the environment variable `OMP_SCHEDULE`. We tried and gained some parallel speed up using `OMP_SCHEDULE=guided,32` and several other `chunk_size` values but the highest gain so far was setting it to `OMP_SCHEDULE=static,70`. As we will discuss later, there is lot of potential to improve on performance by doing experimental design and analysis on the so many factors and levels that we have. It is not hard to see that the performance factors `OMP_SCHEDULE` and block size N_B interact with respect to the Response Time of this hotspot triangularize operation. This is the fastest solution of all, we tried different scenarios to check whether the level-3 BLAS variant would outperform this version it didn't even for very large problem sizes. The extra memory copying and extra flops to accumulate and apply the rotations to the row-block simply defeats the gains of using level-3 BLAS. The performance results of transposing and auto-vectorizing were also disappointing

Listing 4.19: Choice 1 applying Givens rotations directly and OpenMP

```
#pragma omp parallel for schedule(runtime) \
  private(xx0, xx1, xx2, xx3, xx4, xx5, xx6, xx7, xx8, uu0,
      uu1, uu2, uu3, uu4, uu5, uu6, uu7, uu8)
  for (int jj = (j + nb); jj < n; jj++) {
    xx0 = r(i - 1, jj);
    xx1 = r(i    , jj);
    xx2 = r(i + 1, jj);
    xx3 = r(i + 2, jj);
    xx4 = r(i + 3, jj);
    xx5 = r(i + 4, jj);
    xx6 = r(i + 5, jj);
    xx7 = r(i + 6, jj);
    xx8 = r(i + 7, jj);

    uu0 = c0*xx0 + s0*xx1;
    uu1 = c0*xx1 - s0*xx0;

    xx1 = uu1;
    uu1 = c1*xx1 + s1*xx2;
    uu2 = c1*xx2 - s1*xx1;

    xx2 = uu2;
    uu2 = c2*xx2 + s2*xx3;
    uu3 = c2*xx3 - s2*xx2;

    xx3 = uu3;
    uu3 = c3*xx3 + s3*xx4;
    uu4 = c3*xx4 - s3*xx3;

    xx4 = uu4;
    uu4 = c4*xx4 + s4*xx5;
    uu5 = c4*xx5 - s4*xx4;

    xx5  = uu5;
    uu5 = c5*xx5 + s5*xx6;
    uu6 = c5*xx6 - s5*xx5;

    xx6  = uu6;
    uu6 = c6*xx6 + s6*xx7;
    uu7 = c6*xx7 - s6*xx6;

    xx7  = uu7;
    uu7 = c7*xx7 + s7*xx8;
    uu8 = c7*xx8 - s7*xx7;

    r(i - 1, jj) = uu0;
    r(i    , jj) = uu1;
    r(i + 1, jj) = uu2;
    r(i + 2, jj) = uu3;
    r(i + 3, jj) = uu4;
    r(i + 4, jj) = uu5;
    r(i + 5, jj) = uu6;
    r(i + 6, jj) = uu7;
    r(i + 7, jj) = uu8;
}
```

The Choice #2 Register blocking and accumulated rotations level 3 BLAS in points 4.3.4 uses the sines and cosines already precomputed in registers to build an accumulated orthogonal Givens rotation matrix $G_{nb}^T \dots G_1^T I = H$. We created a Matlab code generator based on the Symbolic Math Toolbox [37] that generates the structure of H directly on registers without doing the expensive accumulation via N_B MMM as shown in code listing 4.20, it is of course not so interesting for the $N_B = 4$ case but more to bigger sizes ,

[37] http://www.mathworks.com/products/symbolic/

e.g., the $N_B = 16$ which would be very hard to do manually bug-free. The result implementation we can see in code listing 4.21. We build `gc` from the generated Matlab code that contains the result of accumulating the N_B rotations. We then create a patch corresponding to the row block using Intel MKL `mkl_domatcopy` and finally we apply the accumulated Givens at once to the matrix by multiplying it to the patch and placing the result directly on the main matrix using Intel MKL `cblas_dgemm`. As previously discussed this approach would offer better performance if we weren't constrained by the `cblas_dgemm` function signature to create the extra patch and do the unnecessary extra matrix scalar multiply and matrix matrix sum.

Listing 4.20: Accumulated Givens code generator using Matlab

```matlab
clear all;

% block size
nb = 4;

% defining 0 and 1 as symbols too
sym_0 = sym('0');
sym_1 = sym('1');

c0  = sym('c0');
c1  = sym('c1');
c2  = sym('c2');
c3  = sym('c3');

s0  = sym('s0');
s1  = sym('s1');
s2  = sym('s2');
s3  = sym('s3');

% create H orthogonal matrix using the sin and cos symbols
% filling in the first rotation
I=repmat(sym_0,(nb+1),(nb+1));
for i=1:(nb+1)
    I(i,i)=sym_1;
end
H = I;
H(1:2,1:2) = [c0 s0; -s0 c0];

G = I;
G(2:3,2:3) = [c1 s1; -s1 c1];
H = G*H;

G = I;
```

```
G(3:4,3:4) = [c2 s2; -s2 c2];
H = G*H;

G = I;
G(4:5,4:5) = [c3 s3; -s3 c3];
H = G*H;

% generate the C++ code
for i=1:(nb+1)
    for j=1:(nb+1)
        if H(i, j) ~= sym_0
            fprintf('gc(%d, %d)=%s;\t', i - 1, j - 1, char(H(i
                , j)));
        end
    end
    fprintf('\n');
end
```

Listing 4.21: Choice 2 applying Givens rotations with level 3 BLAS

```
// pre-allocated once
static tsfo_matrix<double> gc(nb + 1, nb + 1, 0.0);
static tsfo_matrix<double> t1(nb + 1, 0);
static const double alpha = 1.0;
static const double beta  = 0.0;

//...

if (n - j - nb > 0) {
 // accumulated Givens orthogonal matrix (generated)
 gc(0,0)=c0;            gc(0,1)=s0;
 gc(1,0)=-c1*s0;        gc(1,1)=c0*c1;
 gc(2,0)=c2*s0*s1;      gc(2,1)=-c0*c2*s1;
 gc(3,0)=-c3*s0*s1*s2;  gc(3,1)=c0*c3*s1*s2;
 gc(4,0)=s0*s1*s2*s3;   gc(4,1)=-c0*s1*s2*s3;

 gc(1,2)=s1;
 gc(2,2)=c1*c2;         gc(2,3)=s2;
 gc(3,2)=-c1*c3*s2;     gc(3,3)=c2*c3;  gc(3,4)=s3;
 gc(4,2)=c1*s2*s3;      gc(4,3)=-c2*s3; gc(4,4)=c3;

 // create temporary row block working patch
 {
  const int mm = nb + 1;
  const int nn = n - j - nb;
  t1.cols(nn);
  const lapack_int lda = r.leading_dim();
  const lapack_int ldb = t1.leading_dim();
  mkl_domatcopy(r.char_order(), 'N', mm, nn, alpha, &r(i - 1,
      j + nb), lda, t1.data(), ldb);
}

// apply the accumulated Givens gc to the matrix row block
// (excluding the diagonal block) at once using dgemm
{
  const int mm = gc.rows();
  const int nn = t1.cols();
  assert(nn > 0);
  const int kk = gc.cols();
  const CBLAS_ORDER b_order = r.blas_order();
  const MKL_INT lda = gc.leading_dim();
  const MKL_INT ldb = t1.leading_dim();
  const MKL_INT ldc = r.leading_dim();
  cblas_dgemm(b_order, CblasNoTrans, CblasNoTrans, mm, nn, kk,
       alpha, gc.data(), lda, t1.data(), ldb, beta, &r(i - 1,
      j + nb), ldc);
}
```

The Choice #3 Register blocking and auto-vectorization in points 4.3.4, creates a transposed temporary patch of the row block and uses auto-vectorization to apply the rotations to the trailing columns. The code in listing 4.22 shows the details for this implementation. We can see the two large memory operations it performs to copy/ transpose a temporary patch from and to the main matrix. Further, there we instruct the Intel compiler to enforce vectorization of the loop at line 418 using `#pragma simd`. Pragma simd is designed to minimize the amount of source code changes needed in order to obtain vectorized code [21]. Compiling the code using the Intel compiler `/opt/intel/composer_xe_2013.0.060/bin/intel64/icpc` in an Intel Core 2 Duo architecture, i.e., only SSE2 and no AVX; and using the options `-vec -report6` to generate verbose auto-vectorization output and `-S` instructs to output the actual assembly, we get the following compiler output:

```
sfo_matrix_tria.h(419): SIMD LOOP WAS VECTORIZED.
```

referring to the loop starting at line 419 in our listing. Furthermore, inspecting the assembly we can verify that the code does get auto-vectorized since we see how the SSE2 registers are being loaded and used, this we can check in listing 4.23 and cross check with the line numbers generated in the assembly as comments to the right, we can see in lines 6 through 10 that the matrix elements are loaded from the matrix into vector registers. We were happy to see we manage to get the code auto-vectorized however, it didn't perform better than the other two previous alternatives.

Listing 4.22: Choice 3 transposition with auto-vectorization

```
// ...
if (n - j - nb > 0) {
  // create temporary working patch for the row block
  int mm = nb + 1;
  int nn = n - j - nb;
  t1.rows(nn);
  t1.cols(mm);
  lapack_int lda = r .leading_dim();
  lapack_int ldb = t1.leading_dim();
  mkl_domatcopy(r.char_order(), 'T', mm, nn, alpha, &r(i - 1,
      j + nb), lda, t1.data(), ldb);

  // provide all necessary hints to the compiler:
  // restrict and 32-byte memory address alignment
  typename sfo_type<T>::aptr32 data = t1.data();

  // auto-vectorize
  #pragma simd
  for (int jj = 0; jj < t1.rows(); jj++) {
    xx0 = data[nn*0 + jj];
    xx1 = data[nn*1 + jj];
    xx2 = data[nn*2 + jj];
    xx3 = data[nn*3 + jj];
    xx4 = data[nn*4 + jj];

    uu0 = c0*xx0 + s0*xx1;
    uu1 = c0*xx1 - s0*xx0;

    xx1 = uu1;
    uu1 = c1*xx1 + s1*xx2;
    uu2 = c1*xx2 - s1*xx1;

    xx2 = uu2;
    uu2 = c2*xx2 + s2*xx3;
    uu3 = c2*xx3 - s2*xx2;

    xx3 = uu3;
    uu3 = c3*xx3 + s3*xx4;
    uu4 = c3*xx4 - s3*xx3;

    data[nn*0 + jj] = uu0;
    data[nn*1 + jj] = uu1;
    data[nn*2 + jj] = uu2;
    data[nn*3 + jj] = uu3;
    data[nn*4 + jj] = uu4;
  }
```

```

    // copy temporary patch back to main matrix
    lda = t1.leading_dim();
    ldb = r .leading_dim();
    mkl_domatcopy(r.char_order(), 'T', nn, mm, alpha, t1.data(),
         lda, &r(i - 1, j + nb), ldb);
}
```

Listing 4.23: Generated assembly auto-vectorized code

```
L..LN52641:
        jbe       L_B221.22     # Prob 10%    #419.29
L..LN52642:
L_B221.19:                      # Preds L_B221.18
    movl      %r13d, (%rsp)     #
    movsd     %xmm4, 696(%rsp)  #
    movsd     %xmm1, 688(%rsp)  #
    movsd     %xmm6, 680(%rsp)  #
    movsd     %xmm7, 672(%rsp)  #
    movsd     %xmm8, 616(%rsp)  #
    movq      152(%rsp), %r11   #
    movq      168(%rsp), %r13   #
    movq      160(%rsp), %r14   #
L..LN52643:
L_B221.20:                      # Preds L_B221.19 L_B221.20
L..LN52644:
        movsd     (%r8,%r11,8), %xmm7         #420.11
L..LN52645:
        movaps    %xmm9, %xmm11               #427.14
L..LN52646:
        movsd     -32(%r14,%r11,8), %xmm8     #421.11
L..LN52647:
        movaps    %xmm10, %xmm13              #427.23
L..LN52648:
        mulsd     %xmm8, %xmm11               #427.14
L..LN52649:
        mulsd     %xmm7, %xmm13               #427.23
L..LN52650:
        mulsd     %xmm10, %xmm8               #426.23
L..LN52651:
        mulsd     %xmm9, %xmm7                #426.14
L..LN52652:
        subsd     %xmm13, %xmm11              #427.23
L..LN52653:
        addsd     %xmm7, %xmm8                #426.23
```

4.4 Cache Analysis

In this section we will discuss the reuse or operational intensity corresponding to our Register blocking with loop unrolling Givens update as part of our final HPSFO implementation and corresponding to Step 4 of Wolfe algorithm 1. Our analysis assumes an architecture as described in Table 6.1 particularly 8-way set associative with a cache line consisting of 8 double values. Furthermore we make the best-case scenario assumption that whenever we touch a memory location, the first element of that memory location will be mapped to the first cache-line element, which is in general not true, but this give us a best-case scenario for the analysis. Note that the worst-case scenario would consist on the first memory location we touch ending in the last cache-line element which would result in one extra cache miss in some cases.

We will first define the operational intensity as [40]:

$$I(n) = \frac{\#\text{ operations}}{\#\text{ off-chip accesses (LLC cache misses)}} \tag{4.28}$$

The total number of operations in our case and assuming 6 operations to generate a rotation for one pair and 6 operations to apply a rotation to a pair, where n is the number of columns in the matrix and j the index of the column deleted:

$$\left(\left\lceil\frac{n-j}{N_B}\right\rceil \times \frac{6}{2}\cdot\left(N_B + \frac{N_B(N_B+3)}{2}\right)\right) + \frac{6}{2}(N_B+1) \times \left(\sum_{k=j}^{\left\lceil\frac{n-j}{N_B}\right\rceil}(n-k-N_B)\right) \tag{4.29}$$

We will focus on how the reuse or operational intensity in Eq. 4.28 changes by affecting the denominator with different values of N_B. For a given N_B block size we can find the number of cache misses, which would be an upper bound to the total LLC cache misses. In general, we analyze the case where the matrix is not small enough to fit entirely in cache.

We first notice that to build our stencil previously presented in Eq. 4.27 and to apply N_B blocked rotations, we need a logical block of dimension $(N_B + 1) \times N_B$. Therefore, the amount of memory accesses for a given N_B with the stencil defined in Eq. 4.27 corresponding to a single diagonal red block is:

$$\sum_{i=1}^{N_B}(i+1) = \frac{N_B(N_B+3)}{2} \tag{4.30}$$

the total amount of memory accesses corresponding to the diagonal red block is:

$$\left\lceil \frac{n-j}{N_B} \right\rceil \times \left(\frac{N_B(N_B+3)}{2} \right) \tag{4.31}$$

and the total amount of memory accesses corresponding to the row green block is:

$$(N_B+1) \times \left(\sum_{k=j}^{\left\lceil \frac{n-j}{N_B} \right\rceil} (n-k-N_B) \right) \tag{4.32}$$

evaluating the formula in Eq. 4.30, particularly for $N_B = 8$ we have 44 accesses out of which 9 are cold or compulsory misses [7] because we have 9 rows for $N_B = 8$. Therefore the miss rate corresponding to conquering the diagonal red block is $\frac{9}{44} \approx 0.2$. Applying the rotations to the green block in Eq. 4.27 we get a miss rate of $\frac{2}{9} \approx 0.22$ i.e. 2 cold misses out of 9 memory accesses. Therefore, the miss rate for $N_B = 8$ results in the following operational intensity:

$$I(n) = \frac{\#\text{ operations}}{0.2 \times \text{Diagonal block accesses} + 0.22 \times \text{Row block accesses}} \tag{4.33}$$

Following the same reasoning we get for $N_B = 7$ a very similar result even though in this case we have one miss less in the diagonal block compared to $N_B = 8$, the miss rate for the diagonal block is $\frac{7}{35} = 0.2$ and for the row block the miss rate is $\frac{1}{8} = 0.125$:

$$I(n) = \frac{\#\text{ operations}}{0.2 \times \text{Diagonal block accesses} + 0.125 \times \text{Row block accesses}} \tag{4.34}$$

Here we see that the operational intensity would increase due to almost halving the miss rate corresponding to the row block but it doesn't improve with respect to the diagonal block accesses. We tested both block sizes and could not see the gain of using $N_B = 7$ instead of $N_B = 8$. Contrary to our expectation and for bigger problem sizes we actually observed a performance degradation using $N_B = 7$. Furthermore, we tested the block size $N_B = 16$ and performance degraded considerably even though the miss rate for this case is slightly better $\frac{7+2*8+3*1}{152} \approx 0.17$ and $\frac{3}{17} = 0.17$, we believe this effect is caused by not having enough registers to hold all the needed temporary data while conquering the diagonal block, i.e., register spilling [7].

Note that increasing the N_B to sizes beyond 16 would simply make performance worse due to what we believe is the result of register spilling. Therefore, we would search the optimal $N_B < 16$. Due to time constraints, we didn't investigate further but analyzing, e.g., $N_B = 13$, we get a promising miss rate of $\frac{7+2*5}{104} \approx 0.16$ and $\frac{2}{13} \approx 0.15$ corresponding to the diagonal and row blocks, respectively. It actually takes a lot of time to implement, test, benchmark and compare the stencil of the handcrafted Register blocking with loop unrolling because we don't have a generator for this. Therefore, we leave as part of future work to try out interesting block sizes like $N_B = 13$.

4.5 Cost Analysis

In this section we will cover the cost analysis corresponding to the final HPSFO kernel implementation. We have left out the cost analysis corresponding to the high-performance Krause version for two main reasons: it is too complex to analyze due to the update "sandwiching" and it is not our best implementation performance-wise. The cost function we will present in this section may be validated using Performance Counters (PC) which we do in our `benchmark` application by enabling PAPI [38]

[38]Performance counters may be off, appearing lower flops than the real value for even more than one order of magnitude.

$$C_{\text{flop}}(\text{Step 1}) = \underbrace{(n \times C_{\text{flop}}(\text{EO})}_{\text{Edmonds Greedy}} + \underbrace{2(n+1)}_{\text{initialization}} + \underbrace{2n}_{x=Sw,\ S_{n\times 1}} \tag{4.35}$$

$$C_{\text{flop}}(\text{Step 2}) = \underbrace{(n \times C_{\text{flop}}(\text{EO})}_{\text{Edmonds Greedy}} + \underbrace{\underbrace{nk}_{R^T r_{k+1} = r_{k+1}} + 2nk + 2k + 8n}_{S \leftarrow S \cup \hat{p},\ \text{update R}} \tag{4.36}$$

$$C_{\text{flop}}(\text{Step 3}) = \underbrace{\underbrace{nk}_{R^T \bar{u} = e} + \underbrace{nk}_{Ru = \bar{u}} + 2n}_{\text{Find the MNP } y \text{ in } \textbf{aff } S} + \underbrace{2nk}_{x=Sw,\ S_{n\times k}} \tag{4.37}$$

$$C_{\text{flop}}(\text{Step 4}) = \underbrace{4n + \underbrace{\frac{(k-j)^2}{2}}_{\text{R delete column update } j}}_{\text{Find and remove “mistake”, point } z} \tag{4.38}$$

The detailed cost in flops for the different steps of the Wolfe algorithm 1 corresponding to our HPSFO MNP kernel implementation is shown in Equations 4.35 through 4.38. Here we have that n is the size of the ground set V and k is the size of the corral and number of columns in S and R. Further, we assume that the cost of the `cblas_dtrsm` operation is nm^2 and n^2m for sides 'L' and 'R' respectively [1] which we use to compute the two solve in Step 3 Eq. 4.37 but note that for the three solve steps we do, our right hand size has always only one column and thus, our solve cost is nk [17]. As pointed out previously, we can see that the cost in flops is dominated by Step 3 Eq. 4.37 and Step 4 Eq. 4.38. In the case of Step 4 the cost depends on what index of the matrix most column deletions happen, and this strongly varies from application to application as we will see in the Experimental results chapter 6, e.g., the Minimum Cut consistently leads to many mistakes of the MNP algorithm and the mistakes tend to be the most costly since it is dominated by column deletions among the first column indexes of the matrix. It is easy to see that if column deletions happen more often at the beginning of the matrix then the cost of Step 4 becomes $\frac{n^2}{2}$ whereas deletions at the end of the matrix have constant cost, in fact, using our high-performance foundation deleting a column at the end of our `tsfo_matrix` only takes to decrease the int value of `m_cols`. The final cost will of course depend on the specific EO for the specific problem at hand, e.g., the Log Determinant application as we will see the in chapter 6 is dominated by the EO function evaluation with cost $O(n^3)$ and $O(n^2)$ corresponding to the non-incremental and incremental EO function evaluation, respectively.

Chapter 5

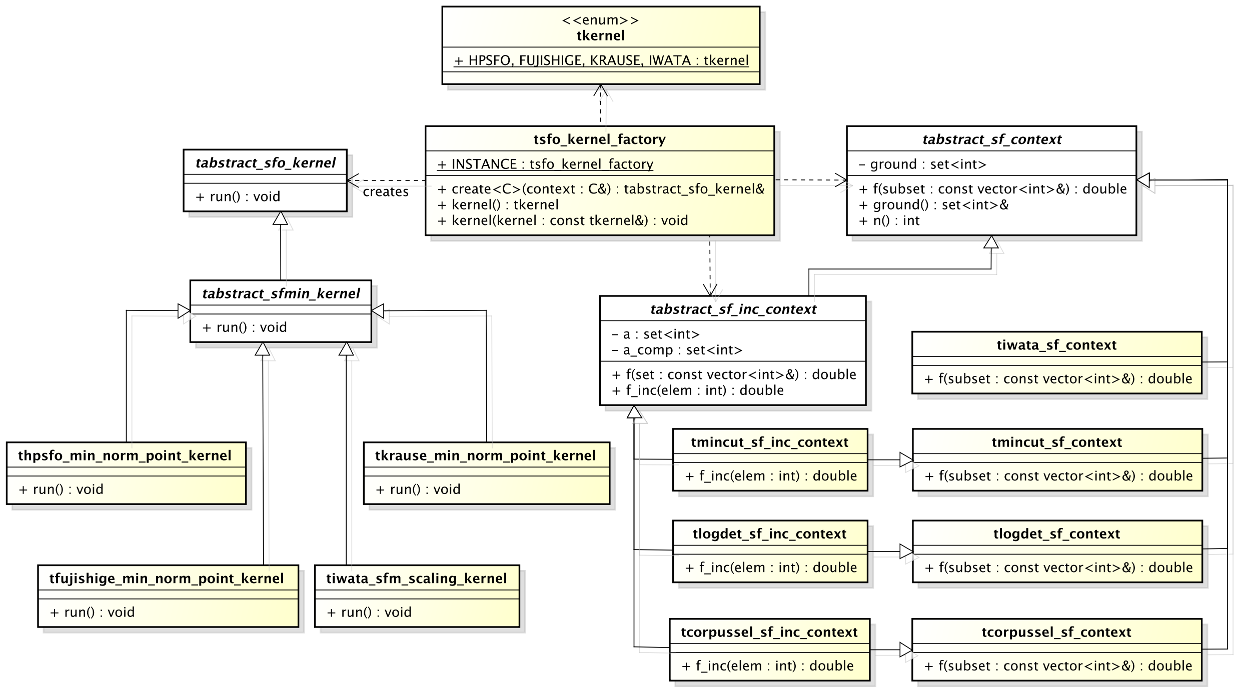
Software Framework

In the Implementation Chapter 4 we discussed the details relevant to optimization and high-performance foundation. In this Chapter we will cover the aspects related to project design, code quality and organization. We strived to the highest quality in all areas not only performance and this is reflected in our Object Oriented (OO) top-down design [35], level of test coverage and ease of extensibility. This Chapter will also serve as a user manual and reference for any future work and extensions to the code base of this work.

5.1 Software Design

The main design of our submodularity framework is depicted in Fig 5.1 [1]. We have two main class hierarchies: the hierarchy corresponding to the SFO algorithm kernels and the hierarchy of the EO function evaluation "contexts". The hierarchy of the SFO kernels only "knows" about the minimal abstract interfaces for the EO function evaluation contexts `abstract_sf_context` and `abstract_sf_inc_context` and this is referred to as the *need-to-know* policy: barring every module from accessing any information that is not strictly required for its proper functioning [35] which allows one to extend both hierarchies with minimal impact on each other. All the SFO kernel needs to know and depend upon is that there are these two abstract types that offer two levels of EO evaluation: non-incremental `f(x)` and incremental `f_inc(x)` and the kernel implementations will try to use the one with the highest capability in this case the incremental version.

[1]We would like to thank Change Vision, Inc. for providing us with a free version of Astah Professional `http://astah.net/` to create this class diagram.



<<enum>>
tkernel
+ HPSFO, FUJISHIGE, KRAUSE, IWATA : tkernel
tsfo_kernel_factory
+ INSTANCE : tsfo_kernel_factory
+ create<C>(context : C&) : tabstract_sfo_kernel&
+ kernel() : tkernel
+ kernel(kernel : const tkernel&) : void
tabstract_sfo_kernel
+ run() : void
creates
tabstract_sf_context
− ground : set<int>
+ f(subset : const vector<int>&) : double
+ ground() : set<int>&
+ n() : int
tabstract_sfmin_kernel
+ run() : void
tabstract_sf_inc_context
− a : set<int>
− a_comp : set<int>
+ f(set : const vector<int>&) : double
+ f_inc(elem : int) : double
tiwata_sf_context
+ f(subset : const vector<int>&) : double
thpsfo_min_norm_point_kernel
+ run() : void
tkrause_min_norm_point_kernel
+ run() : void
tmincut_sf_inc_context
+ f_inc(elem : int) : double
tmincut_sf_context
+ f(subset : const vector<int>&) : double
tlogdet_sf_inc_context
+ f_inc(elem : int) : double
tlogdet_sf_context
+ f(subset : const vector<int>&) : double
tfujishige_min_norm_point_kernel
+ run() : void
tiwata_sfm_scaling_kernel
+ run() : void
tcorpussel_sf_inc_context
+ f_inc(elem : int) : double
tcorpussel_sf_context
+ f(subset : const vector<int>&) : double

Fig. 5.1: Submodularity Framework Class diagram

As we can see on the left side of Fig. 5.1 the SFO kernel hierarchy is quite simple. We have a top abstract type `tabstract_sfo_kernel` which would split into two branches, one for minimization `abstract_sfmin_kernel` and optionally a future possible one for submodular maximization `abstract_sfmax_kernel`. We have of course only covered the branch of submodular function minimization in this work but this design can be easy extended to support submodular maximization kernels as well. The abstract level offers a simple interface to invoke the kernel by simply invoking `run`, initialization is similarly simple and requires the concrete context to optimize over, the ϵ convergence or error and the resolution, which defines the size of the minimum minimizer and is a lower bound for nonzero values of $|f(X) - f(Y)|$ for $Y \subset X$, its default value is 1.0 for integer valued submodular functions [13, 24]. We currently integrated into this design and support three SFM MNP implementations: optimized high-performance Krause implementation, Fujishige-Wolfe base implementation [2] and our high-performance HPSFO fastest MNP kernel implementation. We have thoroughly tested all our kernels, not only validating the correctness of results

[2]Adapted Fujishige base C implementation to the new OO design, added incremental evaluation and duality gap convergence.

but we also validate the number of major and minor iterations for the sole purpose of performance regression testing so we don't introduce code changes that lead to an inflated number of iterations. We have also integrated the Scaling kernel from Iwata [24] but we could not get it to pass the tests beyond the simple Iwata test function evaluation. Iwata's kernel separate from offering lower theoretical complexity bounds i.e., offering lower polynomial complexity achieves so by exhaustively evaluating the EO function a number of times proportional to the ground set size and in practice does not perform well as it gets quickly overtaken by all other kernels even for small problem sizes where this pre-evaluation shouldn't actually take long. [3]

On the right side of Fig. 5.1 we can see the EO function evaluation context hierarchy. Since the EO function evaluation requires a lot more than just a stateless function to evaluate, e.g., in the Minimum Graph Cut case we need the underlying graph *context* or more generally each EO function context needs to provide its own ground set V; we have designed the EO function as a *First-class citizen* [42, 35] and this has greatly simplified the design compared to what we initially had implemented in C during the course project (and same for other C SFM implementations) where the kernels needed to know not only about the function pointers but also about what data to pass to those functions, this design was inherited in our initial C implementation from the Krause submodularity Matlab toolbox [30] and Bach framework uses the same idea [4]. In the case of incremental function evaluation we do really need a lot more context, e.g., efficient set representation to keep track of what elements belong to A and its complement $A^C = \{V \setminus A\}$, etc. We have two main abstractions to choose inheriting from when we want to introduce a new application namely `tabstract_sfo_context` and `tabstract_sfo_inc_context` for non-incremental and incremental evaluation, respectively. Note that if we add support for incremental evaluation, the basic function evaluation must also be available not only for testing purposes [4] but also to evaluate the function as part of the MNP algorithm when finding the norm used to test for convergence while using the duality gap convergence criteria, and therefore we have a multiple inheritance which is not really an issue of duplicated attribute members while using `virtual` inheritance [46]. Therefore, we have the plain EO function contexts inheriting from `tabstract_sf_context` and the incremental EO function contexts inheriting from `tabstract_sf_inc_context` offering the incremental capa-

[3]We could observe this behavior even when the final results were not correct as it didn't pass our tests.

[4]We use the result of the plain function evaluation to test the incremental function evaluation implementations

bility and inheriting from its non-incremental concrete implementation so it offers both function evaluations, e.g., `tlogdet_sf_inc_context` inherits from `tabstract_sf_inc_context` as *specialization* inheritance [35] and inherits from `tlogdet_sf_context` as *implementation* inheritance [35] effectively inheriting from `tabstract_sf_context` twice. Finally, note that the EO function evaluation context does not "know" anything about the existence of the SFO kernel hierarchy and thus, abstract kernels and concrete kernel implementations may evolve without affecting the EO function contexts in any way.

Finally, in the middle of the diagram we see the `tsfo_kernel_factory` whose sole responsibility is to create SFO kernels and therefore, abstract away the "plumbing" of SFO kernels and EO function evaluation contexts. It offers a generic `create` function to that creates ready-to-run SFO kernel instances, and it is generically parameterized with the actual EO function context type being used otherwise we would need two `create` functions [46]. This class is implemented as a Singleton [14] offering a convenient way to instantiate the working instance of the SFO kernel. Putting it all together the sample code depicted in listing 5.1 shows how the different abstractions fit together and are used to run our "two-moons" semi-supervised clustering SFM problem solver with incremental evaluation. Lets break down this code listing into several easy steps, up to line 5 we initialize the submodular function context including parameters specific to the two-moons problem. Line 6 uses `tsfo_kernel_factory` to create a ready-to-run SFM kernel over the Log Determinant context. Line 10 runs the context and from line 13 we obtain the results including the optimal subset and the optimal function evaluation on that subset. Further we can also obtain some details related to the execution of the SFM kernel. Note we don't see anywhere what concrete SFM kernel was used, unless a specific SFO kernel is set, `tsfo_kernel_factory` will use the default one, our favorite and indisputably fastest HPSFO `thpsfo_min_norm_point_kernel` implementation. We can set or change the current preferred kernel to create by invoking the class property writer `tsfo_kernel_factory::INSTANCE.kernel(FUJISHIGE);` in this case we made the Fujishige MNP kernel the default one for SFM. If we wanted to execute the exact same Log Determinant application but without incremental function evaluation (perhaps for the sake of performance comparison), the only change required to listing 5.1 would be at line 5 replacing `tlogdet_sf_inc_context` by `tlogdet_sf_context`, easy enough!

Listing 5.1: Log Determinant "two moons" semi-supervised clustering SFM solver

```
// LogDet requires specific resolution
double resolution = 0.0;

// setup the LogDet application using incremental evaluation
tlogdet_sf_inc_context sf_context(K, s, fv);
tabstract_sfo_kernel& sfo_kernel =
tsfo_kernel_factory::INSTANCE.create(sf_context,
   tabstract_sfo_kernel::DEFAULT_EPSILON, resolution);

// run the kernel, takes long?
sfo_kernel.run();

// retrieve the solution or optimal subset
tsfo_vector<int> subset = sfo_kernel.subset();

// retrieve the optimal value corresponding to the
// optimal subset above
double f_eval = sfo_kernel.subopt();

// find out a bit more about the problem we just solved
long    major_iter  = sfo_kernel.major_iter();
long    minor_iter  = sfo_kernel.minor_iter();
uint_64 flops_count = sfo_kernel.flops_count();
```

How would the situation change if we wanted to solve instead say, e.g., the Minimum Graph Cut via SFM? the code listing 5.2 shows how to do so, which looks strikingly similar to the listing 5.1, and it is no coincidence. Note that the only change we intuitively need, is to change to the specific EO context corresponding to the Minimum Cut problem, the rest remains the same.

Listing 5.2: Minimum Graph Cut SFM solver

```
// setup the MinCut application, requires a graph
tmincut_sf_inc_context sf_context(graph);
tabstract_sfo_kernel& sfo_kernel = tsfo_kernel_factory::
   INSTANCE.create(sf_context);

// run the kernel, takes long?
sfo_kernel.run();

// retrieve the solution or optimal subset
tsfo_vector<int> subset = sfo_kernel.subset();

// retrieve the optimal value corresponding to the
// optimal subset above
double f_eval = sfo_kernel.subopt();
```

```

  // find out a bit more about the problem we just solved
  long     major_iter  = sfo_kernel.major_iter();
  long     minor_iter  = sfo_kernel.minor_iter();
  uint_64 flops_count = sfo_kernel.flops_count();
```

In the two listings discussed before 5.1 and 5.2 we notice that in order to simply use or invoke our SFM framework, the submodularity library client code would need to "know" about our class hierarchies: kernels and contexts; factory, sfo vector and matrix, etc. The dependency chain gets longer and longer to the extreme of Intel MKL, Boost etc. Therefore, we alternatively offer a "zero dependency" API discussed also in section 5.3 with function signatures that have minimal or no dependency other than basic C++ types, e.g., Log Determinant in API or Façade [14] and this is shown in code listing 5.3 which matches exactly the same problem description and parameters as in Bach Matlab submodular package [4]. We took the effort to implement this plain C++ API for every application so that we could easily invoke our SFM solvers from several client code end points and with minimal dependency footprint. We reused these "zero dependency" API functions from different areas: kernels test-suites, benchmarking and for uncomplicated integration with the Roofline Tool [44] to generate Roofline plots for the different applications and kernels.

Listing 5.3: Zero dependency API Façade for the Log Determinant

```
//==========================================================
// Name        : sfo_function_logdet.h
// Author      : Giovanni Azua (azuagarg@student.ethz.ch)
// Since       : 11.05.2012
// Description : Provides the main API entry point for
//               invoking the Log Determinant application
//==========================================================

#ifndef SFO_FUNCTION_LOGDET_H_
#define SFO_FUNCTION_LOGDET_H_

#include <stdint.h>

#include "sfo_kernel_config.h"

/**
 * Solve the two-moons semi-supervised Clustering problem
 * with SFM using incremental evaluation.
 *
 * Input:
 *  @param n size of the problem input
 *  @param K_data kernel matrix (1-D array of dimension n*n)
 *  @param s_data labelled points (1-D array of dimension n)
 *  @param fv Log determinant evaluation on the full set
 *
 * Output:
 *  @param optimal subset
 *  @param optimal_size size of the optimal subset
 *  @param major_iter number of iterations of the major loop
 *  @param minor_iter number of iterations of the minor loop
 *  @param flops_count total number of flops
 */
void sfo_function_logdet(int n, double* K_data, double* s_data
    , double fv, int*& optimal, long& optimal_size, long&
    major_iter, long& minor_iter, uint64_t& flops_count);

/**
 * Solve the two-moons semi-supervised Clustering problem
 * with SFM using non-incremental evaluation.
 *
 * Input:
 *  @param n size of the problem input
 *  @param K_data kernel matrix (1-D array of dimension n*n)
 *  @param s_data labelled points (1-D array of dimension n)
 *  @param fv Log determinant evaluation on the full set
 *
 * Output:
 *  @param optimal subset
 *  @param optimal_size size of the optimal subset
 *  @param major_iter number of iterations of the major loop
 *  @param minor_iter number of iterations of the minor loop
 *  @param flops_count total number of flops
 */
void sfo_function_logdet_noninc(int n, double* K_data, double*
     s_data, double fv, int*& optimal, long& optimal_size,
    long& major_iter, long& minor_iter, uint64_t& flops_count)
    ;

#endif /* SFO_FUNCTION_LOGDET_H_ */
```

5.2 Extensibility guide

This section offers a developers guide to extend the existing framework with new SFM (or in general SFO) kernel implementations or with new applications. The Software design we have provided allows for prescriptive extensibility which can be summarized in a small and easy to follow set of steps.

5.2.1 Adding SFO kernels

Step 1 Extend and implement abstract kernel

The first step to add a new kernel is to implement it by extending the abstract kernel definition i.e., `tabstract_sfmin_kernel`, use existing implementations as example how this is done, e.g., `thpsfo_min_norm_point_kernel` as shown in listing 5.4 we see the typical declarations required and the implementation must also be provided. The initialization of the kernel requires at least the desired Epsilon convergence and Resolution and the initial permutation that could possibly place the starting point closer to convergence. All these parameters have sensible defaults. Special attention must be given to handling the Edmonds Greedy step 2 and 3 since we need at that point to check what capabilities the given context has, and at the end of the algorithm where the size of the minimum minimizer and optimal subset and evaluation must be computed, the existing implementations may be used as example.

Listing 5.4: Adding a new SFO kernel HPSFO example

```
#include "abstract_sfmin_kernel.h"

class thpsfo_min_norm_point_kernel : public virtual
    tabstract_sfmin_kernel {
public:
  // constructor
  thpsfo_min_norm_point_kernel(tabstract_sf_context&
      sf_context, double epsilon = DEFAULT_EPSILON,
       double resolution = DEFAULT_RESOLUTION, tsfo_vector<int>
            initial_perm = tsfo_vector<int>());
  thpsfo_min_norm_point_kernel(tabstract_sf_inc_context&
      sf_inc_context, double epsilon = DEFAULT_EPSILON,
       double resolution = DEFAULT_RESOLUTION, tsfo_vector<int>
            initial_perm = tsfo_vector<int>());

  // run the optimization
  virtual void run();

  // destructor
  virtual ~thpsfo_min_norm_point_kernel();
};
```

Step 2 Add the kernel to factory

For the kernel to be accessible as depicted in our main design 5.1 we need to add it to the `tsfo_kernel_factory` in file `sfo_kernel_factory.h`. Simply add a new constant so the new kernel can be uniquely identified, e.g., code listing 5.5 where we add `XXX` kernel.

Listing 5.5: Add kernel constant identifier

```
enum tkernel {
  KRAUSE, FUJISHIGE, HPSFO, IWATA, /* >>> */ XXX /* <<< */
};
```

Then extend the `tsfo_kernel_factory` implementation to support creating the new kernel, e.g., code listing 5.6.

Listing 5.6: Add kernel creation code to factory

```
template<class C>
inline tabstract_sfo_kernel& tsfo_kernel_factory::create(C&
    context, double epsilon, double resolution, tsfo_vector<
    int> initial_perm) {
  // ...

  switch (m_kernel) {
    // ...
    case XXX: {
      m_sfo_kernel = new txxx_kernel(context, epsilon,
          resolution, initial_perm);
      break;
    }
    // ...
  }

  assert(m_sfo_kernel != NULL);
  return *m_sfo_kernel;
}
```

Step 3 Add the kernel to kernel configuration

Now that we have populated the factory with the new kernel we need to add the ability for the new kernel to be set as factory default from the “zero dependency” API by adding the appropriate function and setter invocation in file `src/api/sfo_kernel_config.h` and provide the implementation in `src/sfo_kernel_config.cc` as shown in example code listing 5.7.

Listing 5.7: Add kernel configuration for new kernel

```

// ==================================
// src/api/sfo_kernel_config.h
// ==================================

/**
 * Enables XXX's kernel
 */
void set_xxx_kernel();

// ==================================
// src/sfo_kernel_config.cc
// ==================================

/**
 * Enables XXX kernel
 */
void set_XXX_kernel() {
  tsfo_kernel_factory::INSTANCE.kernel(XXX);
}
```

At this point, we have successfully integrated the new kernel `XXX` in our design and framework and it can be set as default SFM kernel for benchmarking or testing or for execution from client code and applications or to be reused from Matlab.

Step 4 Make the new kernel available for benchmarking

The benchmarking is implemented in file `src/test/benchmark.cc` and there we use Boost `program_options` library [5] for the flexibility of supporting and handling many application arguments. If we want to add support for our new demonstrative kernel `XXX` we simply first add it as possible choice in the help corresponding to the input argument i.e., `--kernel=xxx`, and add the appropriate invocation to the kernel configuration that we implemented in the previous step. The sample code listing 5.8 demonstrates how to do this.

Listing 5.8: Add kernel configuration for new kernel

```
po::options_description desc("Sfo benchmark options");
desc.add_options()
 // ...
 ("kernel", po::value<string>(&kernel)->default_value("hpsfo")
    , "Sfo kernel e.g. krause, fujishige, hpsfo, iwata, xxx")
 // ...
;
```

[5] `http://www.boost.org/doc/libs/1_51_0/doc/html/program_options.html`

```

po::variables_map vm;
po::store(po::parse_command_line(argc, argv, desc), vm);
po::notify(vm);

if (vm.count("help")) {
  cout << desc << endl;
  return EXIT_SUCCESS;

} else {
  // ...
  } else
  if (kernel == "xxx") {
    // set xxx kernel as current
    set_xxx_kernel();

  } else
  // ...
}
```

Once this is done, that's all it takes for the benchmarking framework to invoke and produce performance results for our new example kernel implementation `XXX`.

Step 5 Add tests for all applications

To test a new kernel we simply add a corresponding new test-case to each existing application test-suite. So far we have four application test suites i.e., three applications plus Iwata test function test-suite. The test-suite files are: `test/sfo_iwata_test.cc`, `test/sfo_mincut_test.cc`, `test/sfo_logdet_test.cc` and `test/sfo_corpussel_test.cc`. Lets take the example of `test/sfo_iwata_test.cc`, code listing 5.9 demonstrate how this is done. Once the test-suite is implemented for each application, it is fairly simple to add a new kernel for testing. We use the googletest framework [6] as test infrastructure. The `TEST` macro has two parameters, first one is the name of the test-suite, in this case `Iwata` and the second corresponds to the name of the test-case in this case `XXX_Iwata` [7]. Note that when we create a new application test-suite, we provide functions, e.g., in this case `iwata_test` to test all kernels. There are some situations however where we will want to customize the test oracle to have the specific tests check for a precise number of expected major and minor iterations, we do this often to make sure that not only the results are

[6] `http://code.google.com/p/googletest/`
[7] `http://code.google.com/p/googletest/wiki/Primer#Simple_Tests`

correct but also have a stricter check, that there wasn't a change leading to diverging number of iterations and thus, result in a negative impact in performance.

Listing 5.9: Adding new kernel to the Iwata test-suite

```

TEST(Iwata, XXX_Iwata)
{
  // set the current kernel
  set_xxx_kernel();

  // invoke the Iwata test
  iwata_test(iwata_min_test_data, iwata_min_num_tests);
}
```

5.2.2 Adding applications

Step 1 Extend and implement abstract function context

To add a new application the first need to implement the abstract EO function context definition `tabstract_sf_context` and most importantly provide the implementation for the virtual function `f(x)`. Here again the best way to start would be to take a look at an existing EO function context implementation, e.g., code listing 5.10 shows the full self-contained example corresponding to the non-incremental Iwata function evaluation. We see at line 14 of this listing the implementation of the constructor that is mainly responsible to initialize the ground set, the ground set is declared in the superclass. Then at line 22 we have the actual function Iwata evaluation implementation.

Listing 5.10: Example EO function context implementation

```
 1 class tiwata_sf_context : public tabstract_sf_context {
 2 public:
 3   // constructor
 4   tiwata_sf_context(int n);
 5 
 6   // function evaluation
 7   virtual double f(const tsfo_vector<int>& x);
 8 
 9   // virtual destructor
10   virtual ~tiwata_sf_context();
11 };
12 
13 // constructor that initializes the ground set
```

```
tiwata_sf_context::tiwata_sf_context(int n) :
    tabstract_sf_context(n) {
  // initialize the ground set
  for (int i = 1; i <= m_n; i++) {
    m_ground.append(i);
  }
}

// function evaluation
double tiwata_sf_context::f(const tsfo_vector<int>& x) {
  int size = x.size();
  double sum = 0;

  for (int i = 0; i < x.size(); i++) {
    sum += 5 * (double) x[i];
    sum -= 2 * m_n;
  }

  return size * (m_n - size) - sum;
}
```

Step 2 [Optionally] Add support for incremental evaluation

Once the non-incremental function evaluation is implemented and tested, its results can be used as test oracle for testing the incremental evaluation version. Once more, it makes sense to use an example to illustrate the idea, e.g., code listing 5.11 shows the definition for the incremental evaluation EO function context corresponding to the Corpus Selection problem. In this listing we have overridden the most important member functions `f_inc` and `reset`. The private members reveal what data it uses to compute the evaluation incrementally, namely it uses `m_vocabulary_set` to keep track of the vocabulary set of X i.e. the distinct set of words corresponding to the utterances that have been chosen by invoking `f_inc` so far and additionally uses `m_weight_sum`[8] to keep track of the total weight sum of the utterances which have not been passed so far and that belong to the set $X^C = \{V \setminus X\}$, this total weight sum is decreased by the edge weight of the utterance being passed and is initialized or reset to $\sum_{u \in U} w_u$ namely the weights corresponding to the edges between the source node and each utterance.

Listing 5.11: Example EO function incremental context implementation

[8]We compute the total sum once and save it in `m_total_weight_sum` and reset the `m_weight_sum` value to the pre-computed total sum.

```
class tcorpussel_sf_inc_context : public virtual
    tabstract_sf_inc_context, public virtual
    tcorpussel_sf_context {
private:
  tbitset m_vocabulary_set;
  double  m_weight_sum;
  double  m_total_weight_sum;

public:
  // constructor
  tcorpussel_sf_inc_context(const thp_adjlist_bidir& graph,
      double lambda = DEFAULT_LAMBDA);

  virtual double f_inc(int x);

  virtual void reset();

  virtual ~tcorpussel_sf_inc_context();
};

// function evaluation
double tcorpussel_sf_inc_context::f_inc(int x) {
  // make sure it is an utterance
  assert(m_ground[0] <= x && x <= m_ground[m_ground.size() -
      1]);

  const sfo_type<int>::aptr32 out_startpos = m_graph.
      out_startpos();
  const sfo_type<int>::aptr32 out_end_nodes = m_graph.
      out_end_nodes();
  const sfo_type<int>::aptr32 in_startpos = m_graph.
      in_startpos();
  const sfo_type<double>::aptr32 in_weights = m_graph.
      in_weights();

  int num_vertices = m_graph.num_vertices();

  // sum the weight of the utterances already in the set
  int idx = in_startpos[x];
  m_weight_sum -= (double) in_weights[idx];

  // update the current vocabulary
  int a_i = x;
  int end_idx;
  if (a_i < num_vertices - 1) {
    end_idx = out_startpos[a_i + 1];
  } else {
    end_idx = m_graph.num_edges();
  }

  int n = m_ground.size();
  for (int j = out_startpos[a_i]; j < end_idx; ++j) {
    int element = out_end_nodes[j] - n - 2;
    m_vocabulary_set.set(element);
  }

  int vocabulary_size = m_vocabulary_set.size();

  double gamma = (double) vocabulary_size;

  return m_weight_sum + m_lambda*gamma;
}

// function reset
void tcorpussel_sf_inc_context::reset() {
  // reset the vocabulary set
  m_vocabulary_set.clear();
  m_weight_sum = m_total_weight_sum;
}
```

Step 3 Implement “zero dependency” API

After we have the EO function evaluation context and optionally its incremental version, we move into creating a “zero dependency” API which we prefer for the sake of simplicity over increasing and scattering the dependency footprint. The zero dependency API consists of a function or set of functions with minimal dependencies that executes the SFM for the new application. We have already discussed the code listing that illustrates this idea in 5.3 and the implementation 5.1. As convention and to keep the code base organized we place the header files containing the declarations of these API functions in the folder, e.g., `src/api/sfo_function_logdet.h` and the implementation in `src/sfo_function_logdet.cc` and the idea is that if there are other libraries or applications that require access to the SFM functionality with minimal dependency, we can do so by adding `src/api` to the includes directory path and linking against the compiled version of our library. We do this already to integrate with the Roofline tool [44].

Step 4 Make the new application available for benchmarking

Once we have the new application integrated, we can start running benchmarks to compare how well the different SFM kernels perform with respect to this new application. We need to modify the implementation file `test/benchmark.cc` and add a static function wrapper which invokes the “zero dependency" API and is passed to the benchmarking framework for automated benchmarking i.e. that execute this function wrapper multiple times and measures elapsed times, taking means and standard deviations etc. An example wrapper function is shown in code listing 5.12 corresponding to the Log Determinant application with and without incremental evaluation. The two most important points while implementing this wrapper is first to keep the wrapper signature unchanged and second the wrapper should only invoke the application SFM solver and not do things like loading application-specific parameters from disk, e.g., graph in DIMACS format [5] etc.

Listing 5.12: Integrating new application with benchmark, wrapper function

```
static void logdet_workload_wrapper(long &minor_iter, long &
    major_iter, uint64_t &flops_count) {
  // invoke the kernel API
  int *optimal = NULL;
  long optimal_size;
  sfo_function_logdet(n, K_data, s_data, fv,
      optimal, optimal_size, minor_iter, major_iter,
          flops_count);
  delete[] optimal;
}

static void logdet_workload_wrapper_noninc(long &minor_iter,
    long &major_iter, uint64_t &flops_count) {
  // invoke the kernel API
  int *optimal = NULL;
  long optimal_size;
  sfo_function_logdet_noninc(n, K_data, s_data, fv,
      optimal, optimal_size, minor_iter, major_iter,
          flops_count);
  delete[] optimal;
}
```

We need to also modify the `main` function in `test/benchmark.cc` to handle the new application, example listing 5.13 demonstrates this again for the case of the Log Determinant application. The "benchmark framework" wasn't actually meant to support accommodating hundreds of applications but a few, to evaluate the performance differences between the implemented kernels and for a handful of applications. The benchmark file can be easily extended to a framework, in fact its core function `run_benchmark` remains unchanged when adding not only new SFM kernel implementations but also the EO function contexts corresponding to new applications.

Listing 5.13: Integrating new application with benchmark

```
if (workload == "logdet") {
  if (input_file.empty()) {
    cerr << "Input Error: 'input-file' must be provided.\n";
    return EXIT_FAILURE;
  }

  // read application problem file
  const char* filename = input_file.c_str();
  ifstream is(filename);
  tlogdet_moons_generator generator;
  is >> generator;

  // application parameters
  n      = generator.n();
  K_data = generator.K();
  s_data = generator.s();
  fv     = generator.fv();

  // run benchmark
  if (INCEVAL) {
    run_benchmark(logdet_workload_wrapper);

  } else {
    run_benchmark(logdet_workload_wrapper_noninc);
  }

  // ...
```

Step 5 Add application test-suite

For each supported application we have one test-suite containing three types of tests: test the function evaluation non-incrementally, test the function evaluation incrementally (both should produce the same result), and test the different SFM kernel implementations against the incremental function evaluation by invoking the appropriate "zero dependency" API functions. We have built a test support class `set_factory.h` that given an interval of integer possible values generates the following sets: empty, full, singleton first, singleton last, odd and even. The function evaluation and kernel tests use this `set_factory.h` to test each application using the different generated sets.

5.3 Project Structure

The project is structured as shown in Fig. 5.2 which matches the typical structure for projects when using Subversion. We have the root `$SFO_HOME` where the project sources are, next we have `tags` where we branch out the different releases and we have `trunk` where we have the current working version. Under `trunk` we have `code` where the sources are and at the same level `build` and `build_debug` corresponding to Release and Debug builds respectively, these are generated using CMake build framework [9] as will be discussed in section 5.4. Under `code` we find `src` where the sources of our library are and `test` containing all test-suites and test infrastructure code, e.g., `benchmark.cc` and `set_factory.h` implementations. The CMake project file CMakeLists.txt is found under `code` and will be described in the section 5.4. The `src` folder contains several sub-folders as described in points 5.3. Currently we have over 60 test-cases corresponding to SFM kernels, applications, high-performance code foundation matrix and vector, and more.

api Contains only header files corresponding to the "zero dependency" API with minimal dependency footprint other than C++ primitive types in order to invoke our SFM framework functionality, e.g., SFM solve Minimum Graph Cut, Log Determinant, Iwata or Corpus Selection. We use Boost BGL [10] to load the graph files in DIMACS format [5] and this is also made part of the `api` folder. In conclusion, any client application wanting to use our SFM solvers for the implemented applications should include this `api` folder in their include folder list, e.g., in gcc this is done using the environment variable `C_INCLUDE_PATH` or `CPLUS_INCLUDE_PATH` [11].

context Contains the abstract EO function contexts header and implementation files for all applications with non-incremental and incremental variations.

fortran Contains a few Fortran routines reused for the QR updates, e.g., Kressner routine [12] for updating a QR decomposition after appending a row block [32].

kernel Contains abstract and concrete SFM kernel implementations: Fujishige, Krause, Iwata and HPSFO.

[9] `http://www.cmake.org/`
[10] `http://www.boost.org/doc/libs/1_51_0/libs/graph/doc/index.html`
[11] `http://gcc.gnu.org/onlinedocs/gcc/Environment-Variables.html`
[12] `http://www.math.ethz.ch/~kressner/qrupdate.php`

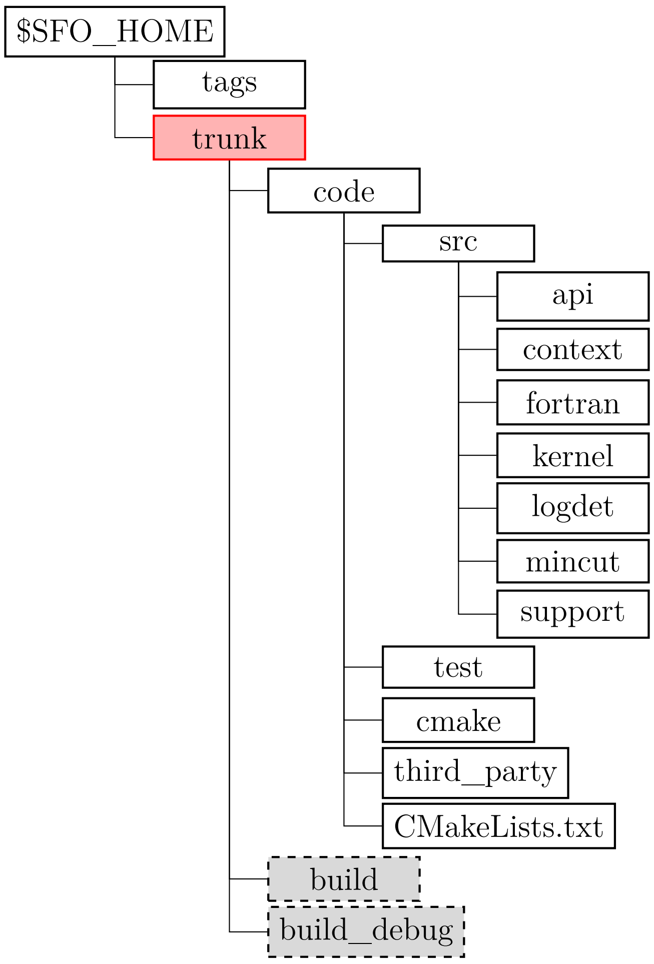


Fig. 5.2: Submodularity project structure

logdet Implementation files specific to the Log Determinant application, here we included the file `logdet_moons_generator.cc` but the name is a bit misleading, this implementation was initially intended to mirror Bach's Matlab implementation for generating two-moons data sets but was instead written to serialize and deserialize the two-moons dataset files exported from Matlab. The reason for not generating the two-moons data was two fold, first to avoid spending too much time on it, e.g., porting to C++ some Matlab operations etc and second to have the exact same input as the Bach's MNP algorithm does i.e., generating the files involves using random generators etc.

mincut Contains a standalone application for executing the Minimum Cut SFM solver via the "zero dependency" API. This we use among others, as part of the benchmark scripts to test or validate the solutions produced during benchmarking i.e., make sure that the benchmark results are correct.

support Contains support or helper implementations for the submodularity framework, e.g., `bitset.h` a binary set representation with constant time for adding an element to the set and testing whether an element belongs to a set. There we also have `bufferpool.h`, `quicksort.cc`, `linear_sort.h` for sorting integer-valued arrays i.e. ground set index elements in $O(n)$ time, etc.

5.4 Build Process

We employed the CMake project for building and generating project files, e.g., Eclipse, Visual Studio or Xcode project files. The project is configured via the `CMakeLists.txt` file [34]. In this CMake project specification file we mainly define: external dependencies including their include and link paths, toggle project-wide macros, define compilation parameters and settings, define project targets. So far we have the following project targets: submodularity library, benchmark standalone application, mincut standalone application, all test suites using googletest and the Matlab MEX library to use HPSFO directly from Matlab, we will discuss this in details in section 5.6.

The file `README.txt` contains the most important commands we used to, e.g., generate project files, generate build directories in Release and Debug modes, use `valgrind` to troubleshoot possible segmentation faults and in general work with our submodularity framework. For completeness we are going to include some of the most important commands here.

The following command generates and executes the Release build:

```
$ cd $SFO_HOME/trunk
$ rm -rf build; mkdir build; cd build;
$ cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_C_COMPILER=icc \
        -DCMAKE_CXX_COMPILER=icpc \
        -DCMAKE_Fortran_COMPILER=ifort ../code
$ make
```

The following command generates and executes the Debug build:

```
$ cd $SFO_HOME/trunk
$ rm -rf build_debug; mkdir build_debug; cd build_debug;
$ cmake -DCMAKE_BUILD_TYPE=Debug -DCMAKE_C_COMPILER=icc \
        -DCMAKE_CXX_COMPILER=icpc \
        -DCMAKE_Fortran_COMPILER=ifort ../code
$ make
```

The following command generates Eclipse project files:

```
$ cd $SFO_HOME/trunk/build_debug
$ cmake -G "Eclipse CDT4 - Unix Makefiles" ../code
```

5.5 Software Dependencies

The submodularity framework we implemented has the following main software dependencies:

Boost We use the following Boost libraries: `filesystem`, `system`, `graph` , `program_options` and `chrono`. We use the Graph library to load maxflow DIMACS graph files and to find the Minimum Graph Cut using Edmonds Karp implementation. We use Boost Chrono library portable high-resolution timer to collect response time for the execution of the SFM implementations. The library `program_options` helps simplify development of standalone applications that have many arguments, e.g., the benchmark standalone application.

PAPI We use the PAPI library [13] to collect performance indicators, e.g., response time, flop counts and mega flops. Performance counters (PC) help validating the estimated theoretical flop count.

Intel MKL As previously discussed, we use Intel MKL as part of our high-performance foundation that provides implementation for BLAS, LAPACK routines and more.

Intel TBB We employed Intel Threading Building Blocks[14] (TBB) to take advantage of parallel sorting as part of the Edmonds Greedy step in both variations non-incremental 2 and incremental 3. We have added as future work the possibility to extend the use of TBB enriching our high-performance foundation even further, e.g., parallelizing loops that can not be made so with auto-vectorization and OpenMP or reusing fast parallel reductions.

[13]`http://icl.cs.utk.edu/papi/`
[14]`http://threadingbuildingblocks.org/`

5.6 Matlab integration

We have built an adapter that allows invoking our HPSFO implementation directly from Matlab [15]. The C++ entry point to Matlab is implemented in `src/sfo_matlab_adapter.cc` [16] where we define the standard Matlab entry point C++ function `mexFunction` with predefined signature. The code listing 5.14 shows our Matlab C++ entry point which is very similar to previous listings e.g, 5.1, it defines a EO function context, creates and run the default SFM kernel and finally returns the optimal subset result to Matlab. The main difference with other API implementations is that the EO evaluation context we use is `tmatlab_sf_adapter`. That is, we have a context that simply wraps a true context defined elsewhere (in this case in Matlab) and surrogates that true context within our framework, but the calls to the EO function evaluation `f(x)` are delegated to the actual Matlab function. Every time the SFM kernel invokes the function, it is calling our `tmatlab_sf_adapter` function implementation which in turn delegates the call to the Matlab function handle passing the context "param" defined also in Matlab, we then return to the SFM algorithm the result of the Matlab function handle invocation.

Listing 5.14: Matlab adapter C++ entry point

```cpp
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
    mxArray *prhs[]) {
  // first argument should be a function handle
  if (!mxIsClass(prhs[0], "function_handle")) {
    mexErrMsgTxt("ERROR: first input argument must be a
        function handle.");
  }

  // second argument should be a struct 'param' context
  if (!mxIsStruct(prhs[1])) {
    mexErrMsgTxt("ERROR: second argument must be a struct.");
  }

  // setup the Matlab context adapter and run the SFM kernel
  tmatlab_sf_adapter sf_context(prhs[0], prhs[1]);
  tabstract_sfo_kernel& sfo_kernel = tsfo_kernel_factory::
      INSTANCE.create(sf_context);
  sfo_kernel.run();

  // output the optimal subset
  tsfo_vector<int> subset = sfo_kernel.subset();
```

[15] We have tested this with Matlab 64-bit R2012a (7.14.0.739)
[16] `http://www.mathworks.com/help/techdoc/apiref/mexfunction.html`

```
  mxArray *result = mxCreateDoubleMatrix(subset.size(), 1,
      mxREAL);
  for (int i = 0; i < subset.size(); ++i) {
    (mxGetPr(result))[i] = subset[i];
  }
  plhs[0] = result;
}
```

In `CMakeLists.txt` we included an additional target that generates a Matlab library corresponding to the C++ entry point in listing 5.14, the name of the generated library is `hpsfo_matlab.mexmaci64` in this case for Mac OS X 64-bit. Starting Matlab and pointing its default folder to the location where our `hpsfo_matlab.mexmaci64` is, we can test the HPSFO from Matlab as shown in the Matlab code listing 5.15 where we define the Iwata test function `F_iwata`, invoke our HPSFO kernel and obtain the optimal subset results.

Listing 5.15: Matlab Iwata example that invokes HPSFO

```
% ==================================
% test integration
% ==================================

clear all;

% define the true context
param.n = 28;
param.ground = 1:28;

% define the Iwata function handler
F_iwata = @(param, A) length(A)*(param.n-length(A))-sum(5*A-2*
    param.n);

% invoke the HPSFO kernel
subset = hpsfo_matlab(F_iwata, param)

% output
subset =

    10
    11
    12
    13
    14
    15
    16
    17
    18
```

```
19
20
21
22
23
24
25
26
27
28
```

Chapter 6

Experimental Results

In this chapter we will discuss the results we obtained comparing our best HPSFO implementation against all others and for the different applications. We are also going to discuss the different aspects that affect performance and how they differ from one workload application to another. Additionally, we will also see that the performance our fastest implementation is lead by multiple factors and levels that interact and therefore we propose an Experimental Design study [27] and Workload characterization [27] to gain the best performance for the specific application workload.

As discussed before we built on top of Intel Parallel Studio [1] and Intel MKL [2] and therefore we also employed the Intel C/C++ Compiler [3]. We compiled the code using the following flags, this we generate via our CMake project build by actually executing `make VERBOSE=1` [34]:

```
$ /opt/intel/composer_xe_2013.0.060/bin/intel64/icpc -fasm-blocks
-pthread -Wall -Wcheck -O3 -DNDEBUG -align -finline-functions
-malign-double -no-prec-div -openmp -complex-limited-range
-xHost -opt-multi-version-aggressive -scalar-rep
-unroll-aggressive -vec-report6 -restrict
-o build/src/kernel/hpsfo_min_norm_point_kernel.cc.o
-c code/fastcode_project/code/src/kernel/hpsfo_min_norm_point_kernel.cc
```

The flags are actually specified per compiler and in the file `$SFO_HOME/code/cmake/compiler_settings.cmake`. These settings are described in detail in the Intel Compiler Users Manual [22] but we will comment on some of

[1]`http://software.intel.com/en-us/intel-parallel-studio-home/`
[2]`http://software.intel.com/en-us/intel-mkl`
[3]`http://software.intel.com/en-us/intel-compilers/`

the most important ones. We employed `-xHost` to gain the best performance for the underlying architecture i.e. generate instructions for the highest instruction set and processor available on the compilation host, use AVX or SSE2 if applies [22]. The `-O3` includes `-O2` which enables auto-vectorization and `-O3` enables more aggressive optimizations, such as prefetching, scalar replacement, and loop and memory access transformations [22]. Finally we enabled `-restrict` to hint the compiler for more aggressive auto-vectorization and `-vec-report6` to show verbose diagnostics on what loops were auto-vectorized and most importantly the reasons why some loops where not auto-vectorized.

While using Intel MKL and from version 11 we have several key MKL configuration environment variables that affect performance. The most important one is the one that controls MKL multi-threading `MKL_NUM_THREADS`, if it is set, it will restrict the number of threads used by MKL to the value given; setting it two 1 will disable parallelism, if it is not set, then MKL will use as many threads as CPU cores available. The other critically important environment configuration is Conditional Numerical Reproducibility (CBWR) `MKL_CBWR` [4] that offers a trade-off between highest performance for the given platform and reproducibility of the results. The other two main configuration parameters we modify via environment variables is the OpenMP ones. `OMP_NUM_THREADS` restricts the number of threads used for OpenMP, setting it two 1 will also disable parallelism. `OMP_SCHEDULE` configures the OpenMP scheduler e.g. `static,70` meaning all the threads are allocated the number of iterations before they execute the loop iterations. The iterations are divided among threads equally by default but in this case we fix it to an optimal level of 70. We conducted most of our experiments disabling parallelism i.e., setting `export MKL_NUM_THREADS=1` and `export OMP_NUM_THREADS=1` we specially do so for the Gflop/s plots. Those experiment for which parallelism is enabled are stated clearly or suffixed with `_mt` meaning multi-threading.

6.1 Benchmark environment

The development and benchmarking of our submodularity framework was done in two platforms. Most of the development was done in a Mac OS X Intel Core Duo architecture depicted in table 6.2. We did most of the benchmarking in an Intel Sandy-Bridge architecture with specification detailed in

[4]`http://software.intel.com/en-us/articles/introduction-to-the-conditional-numerical-reproducibility-cnr`

CPU-core manufacturer	Intel
Model name	Sandy Bridge-E i7-3930K C2
Number of CPU cores	6
CPU-core frequency	3.20Ghz
Max Turbo Frequency	3.80Ghz
Instruction Set Extensions	SSE4.2, AVX
Cycles for FP additions (Latency/Throughput)	(3/1)
Cycles for FP multiplications (Latency/Throughput)	(5/1)
Maximum theoretical FP peak performance (in Gflop/s)	24
Cache Line Size	64-byte
L1 Data (per core)	32kB 8-way set associative
L1 Instruction (per core)	32kB 8-way set associative
L2 Unified (common)	256kB 8-way set associative
L3	12MB 16-way set associative
FP Registers (per core)	16

Tab. 6.1: Sandy Bridge-E CPU specification

CPU-core manufacturer	Intel
Model name	Core 2 Duo T9900
Number of CPU cores	2
CPU-core frequency	3.06Ghz
Instruction Set Extensions	-
Cycles for FP additions (Latency/Throughput)	(3/1)
Cycles for FP multiplications (Latency/Throughput)	(5/1)
Maximum theoretical FP peak performance (in Gflop/s)	6
L1 Data (per core)	32kB 8-way set associative
L1 Instruction (per core)	32kB 8-way set associative
L2 Unified (common)	4MB 8-way set associative
FP Registers (per core)	16

Tab. 6.2: Intel Core 2 Duo CPU specification

table 6.1 [20], as part of a custom Desktop build tailored for conducting the benchmarks of this thesis work 6.3, e.g., we used the Desktop high-end fastest quad-channel RAM available in the market.

We integrated our submodularity framework and generated the Roofline plots with the Roofline tool [44] that currently only works for the Intel Core Duo platform. Due to this constraint we installed dual-boot in the development MacBook Pro using the same OS version Ubuntu 11.10 kernel 3.0.0-23-generic. At this time the Roofline tool can't accurately compute the Operational Intensity due to not being able to accurately measure the transfer rate between Last Level Cache (LLC) and the RAM for the Sandy-Bridge architecture. Therefore our standard performance benchmarks were conducted using the Sandy-Bridge CPU architecture which is AVX-capable and the Roofline plots in the Intel Core Duo architecture which is SSE2-capable only.

All performance benchmarks shown in the next sections were obtained by executing 3 warm up runs and averaging (mean) over 10 repetitions. We also computed standard deviations but they were too small to show i.e. 2 orders of magnitude or more smaller than the mean and therefore not really useful. We implemented several Bourne Shell scripts for running performance benchmarks `$SFO_HOME/code/benchmark_mincut.sh`, `$SFO_HOME/code/benchmark_logdet.sh` and `$SFO_HOME/code/benchmark_corpussel.sh`

These Shell scripts only require successfully completing a Release build and they will invoke the standalone application `benchmark` passing the appropriate parameters and problem files found under root folder `$SFO_HOME/code/test` and sub-folders `genrmf_data`, `genmoons_data` and `genbipartite_data`

Motherboard	Asus Rampage IV Extreme LGA2011
CPU	Intel i7 3930K C2
CPU Cooler	Noctua NH-D14
RAM	Corsair Dominator GT CMT16GX3M4X2133C9 16GB
Graphic Card	EVGA 670 FTW
PSU	Corsair AX1200 Gold
Case	Corsair 800D
Hard drive	240GB Mercury EXTREME Pro 3G SSD
OS	Ubuntu 11.10
OS kernel version	3.0.0-23-generic

Tab. 6.3: Benchmark hardware

respectively.

6.2 Minimum Graph Cut

In this section we will discuss the experimental results corresponding to the Minimum Graph Cut application discussed in section 3.1. We generated several Minimum Graph Cut problems using the GENRMF tool [5] [5] and combined a balanced mix of Wide and Long graphs.

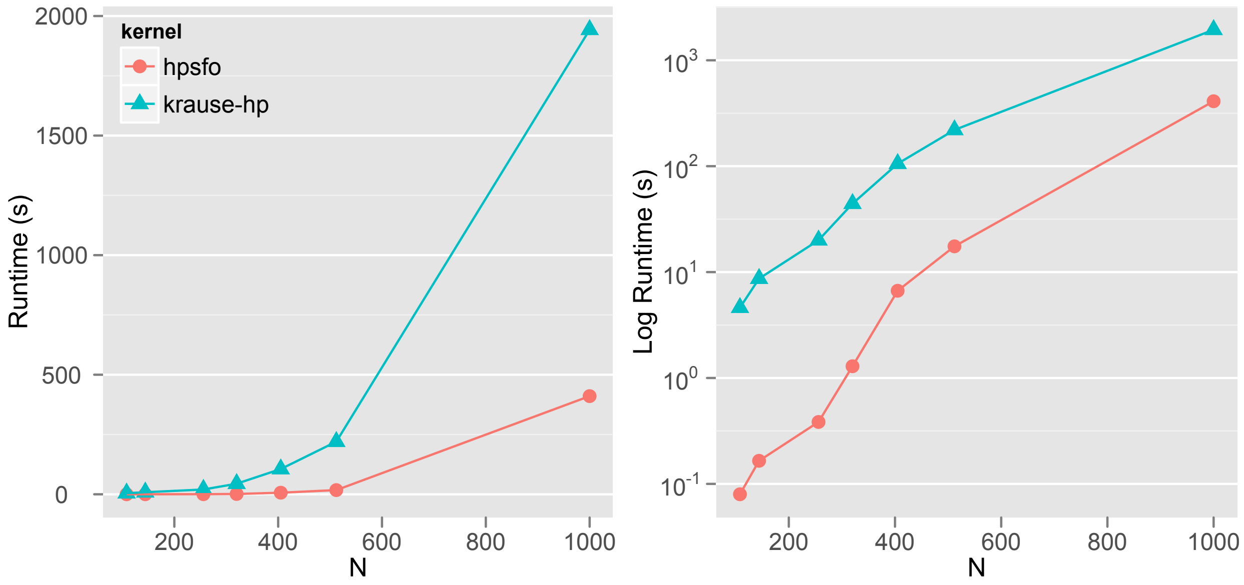


Fig. 6.1: Runtime(s) Min Cut Krause Matlab Toolbox vs HP

We are going to quickly review the performance gains of our high-performance version of Krause MNP kernel implementation. The Fig. 6.1 shows the performance gain of our high-performance version of Krause MNP kernel implementation compared to the original Matlab Toolbox implementation [6]. In this case they both feature incremental function evaluation so we are looking at the gains due to all performance optimizations discussed in sections 4.1 and 4.3.4. We can observe that the performance gap reduces with bigger

[5]`http://www.informatik.uni-trier.de/~naeher/Professur/research/generators/maxflow/genrmf/index.html`
[6]`http://users.cms.caltech.edu/~krausea/sfo/`

problem sizes and our explanation for this is the trade-off between lower flop count and higher Level BLAS. Krause's toolbox recomputes QR each time which belongs to Level-3 BLAS at the cost of cubic flop cost whereas our high-performance version applies increasingly costly Level-2 BLAS updates e.g., generating and selectively applying a set of Householder reflectors in algorithm 5 combining `dlarfg` and `dlarfx`.

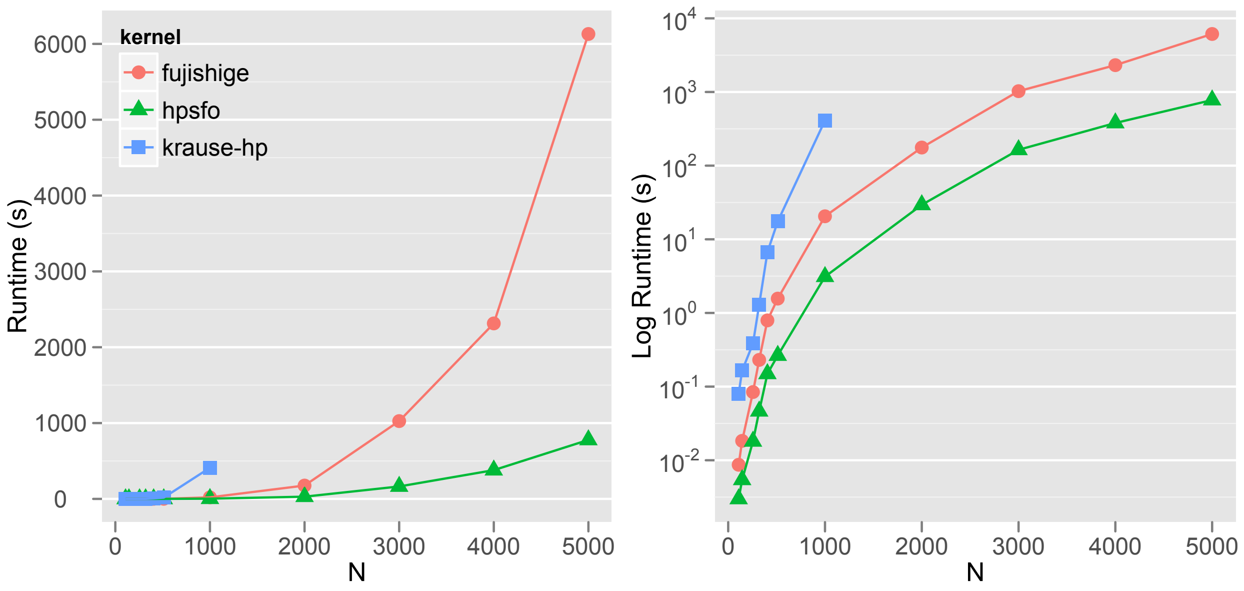


Fig. 6.2: Runtime(s) Min Cut with Incremental Evaluation

Now we will focus only on the performance results for our fastest MNP kernel implementation the HPSFO. Fig. 6.2 depicts the runtime or response time differences between the MNP kernels and using incremental evaluation: Fujishige, Krause and our final HPSFO kernel. We enabled the incremental evaluation feature in the Fujishige implementation to find the net gains due to our faster high-performance implementation. We can observe that our HPSFO implementation outperforms all others taking into account the high-performance foundation, fast memory operations and fast orthogonal updates using Registers blocking with loop unrolling for re-triagularizing the matrix R after a column vector point deletion.

Fig 6.3 shows the gain in our HPSFO kernel implementation due only to incremental EO function evaluation. The gain here is only due to the algorithmically superior implementation corresponding to the modified Edmonds

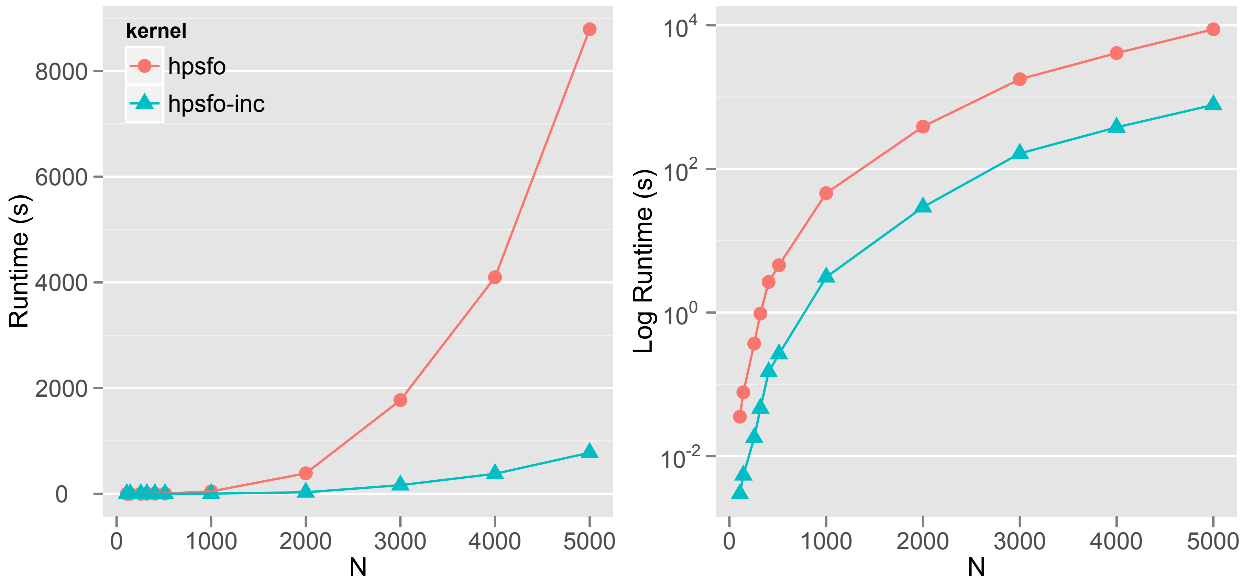


Fig. 6.3: Runtime(s) Min Cut HPSFO: Incremental vs non Incremental

3 where we only need to look at a specific node and its adjacency rather than redoing so for all nodes at every EO evaluation as part of the Greedy Step.

Fig 6.4 shows the performance plot using Gflop/s defined as:

$$\text{Gflop/s} = \frac{\text{flop count}}{\text{Runtime(secs)} \cdot 1e^9} \tag{6.1}$$

Here we fix the flop count to one of the two cases i.e., take the one with highest, and assume they are asymptotically the same and therefore comparable. Note that the flop count for Fujishige and HPSFO is nearly exactly the same except for a few exceptions e.g., as part of Step 4 of Wolfe algorithm 1 we need to repair the broken upper-triangular matrix R, in the HPSFO implementation we generate the sins and cosines once and apply them as shown in listing 4.16 requiring a total of 6 flop to apply the rotation for two row elements per column. Fujishige implementation requires 9 flop to do the same. We can see that for about the same amount of work reflected in flop count we manage to do the same much faster reaching in this up to 90% of scalar peak performance for the Sandy-Bridge architecture described in table 6.1.

The Fig. 6.5 shows the Roofline plot obtained in our Intel Core Duo ar-

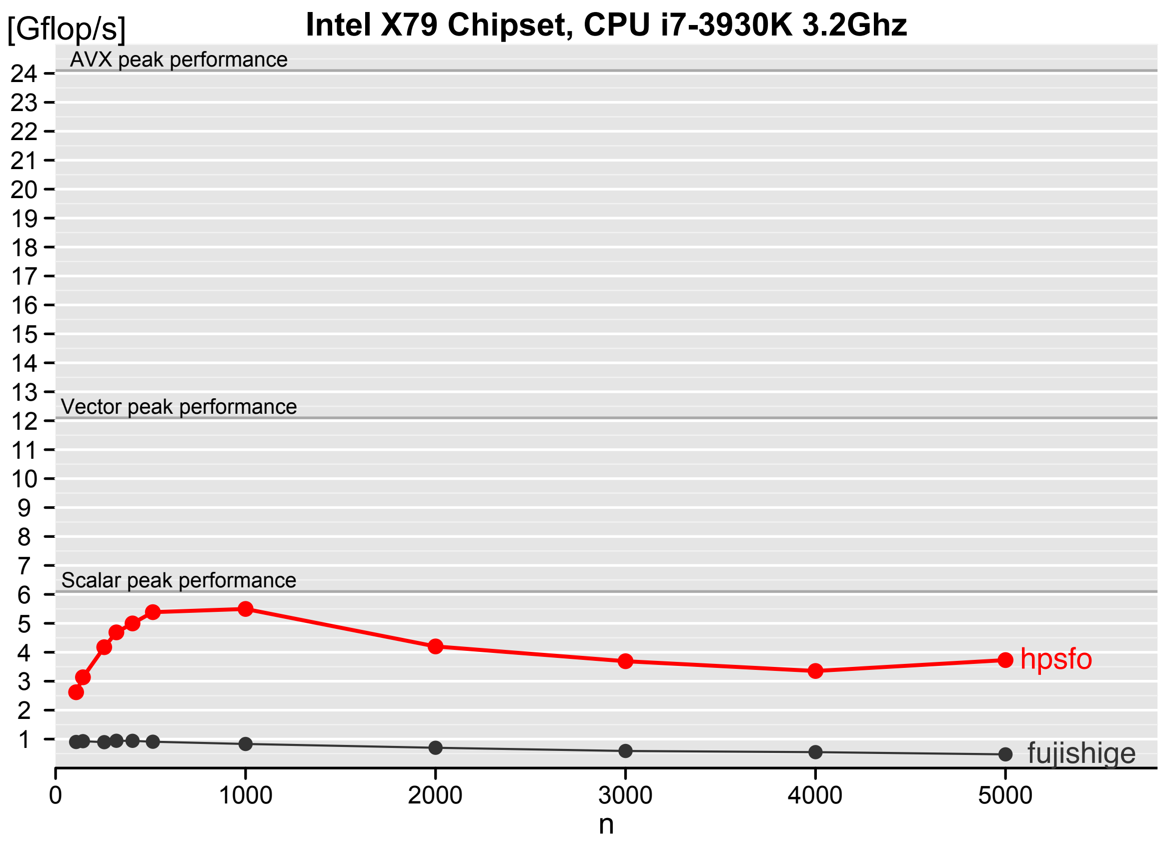


Fig. 6.4: Gflop/s Minimum Cut with inc. evaluation HPSFO vs Fujishige

chitecture 6.2. Here we see that Krause MNP implementation appears to have higher performance Flops but in reality Krause algorithm requires algorithmically a lot more floating point operations to achieve the same therefore inflating the Flops plots. It is therefore not really comparable to the other two MNP kernel implementations Fujishige and HPSFO. However, we can see a similar trend shared across all MNP implementations, the Operational Intensity decreases with bigger problem sizes, meaning that the MNP algorithm is fundamentally memory bound [40]. This effect is what we believe an intrinsic lack of reuse issue of the MNP algorithm where the new points explored by are dynamically and sequentially generated and not often reused for computation. We were nevertheless happy to see that our HPSFO implementation has at least the same Operational Intensity as the Fujishige implementation despite the expensive physical memory operations corresponding to the physical matrix updates due to Step 4 of the Wolfe algorithm 1. Finally, the

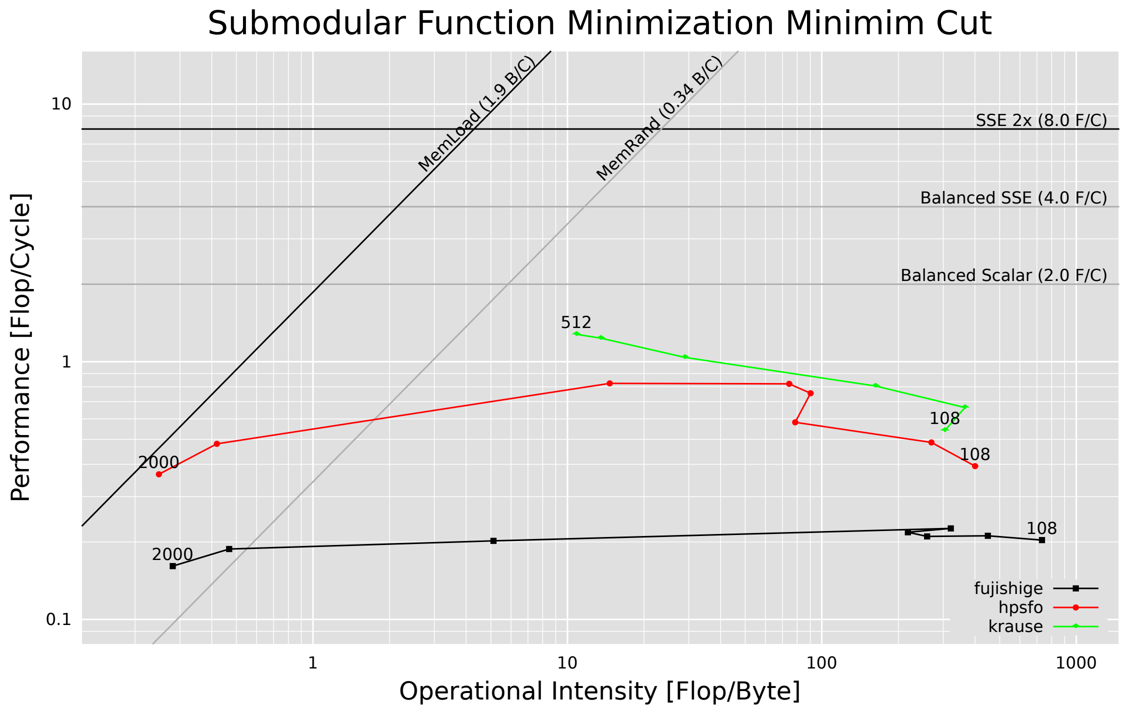


Fig. 6.5: Roofline Minimum Cut with inc. evaluation

Roofline plot confirms the results of our previous Gflop performance plot, that our HPSFO implementation features higher performance than Fujishige for SSE2 as well as AVX-enabled architectures.

The Fig. 6.6 shows the time distribution per Step corresponding to the Steps of Wolfe algorithm 1. This plot give us essentially a dissection of the algorithm revealing where the percentage of time is spent for each case. We have included Fujishige, HPSFO and HPSFO with multi-threading. We can see that in Fujishige MNP implementation, the Step 3 of Wolfe algorithm increasingly becomes the bottleneck with bigger problem sizes, this is due to what we believe the bad locality of Fujishige implementation rooted in the arbitrary column accesses resulting from the high indirection while iterating columns via the double linked list of column indexes. The HPSFO implementation on the other hand, benefits from the solve steps using the highly-optimized Intel MKL implementation of `cblas_dtrsm` on top of page and AVX-aligned contiguous memory. However, we clearly note for HPSFO the higher percentage of computation time corresponding to Step 4 where the physical column deletions take place. Finally, the HPSFO parallel implementation better balances all Steps by executing the re-triangularization of

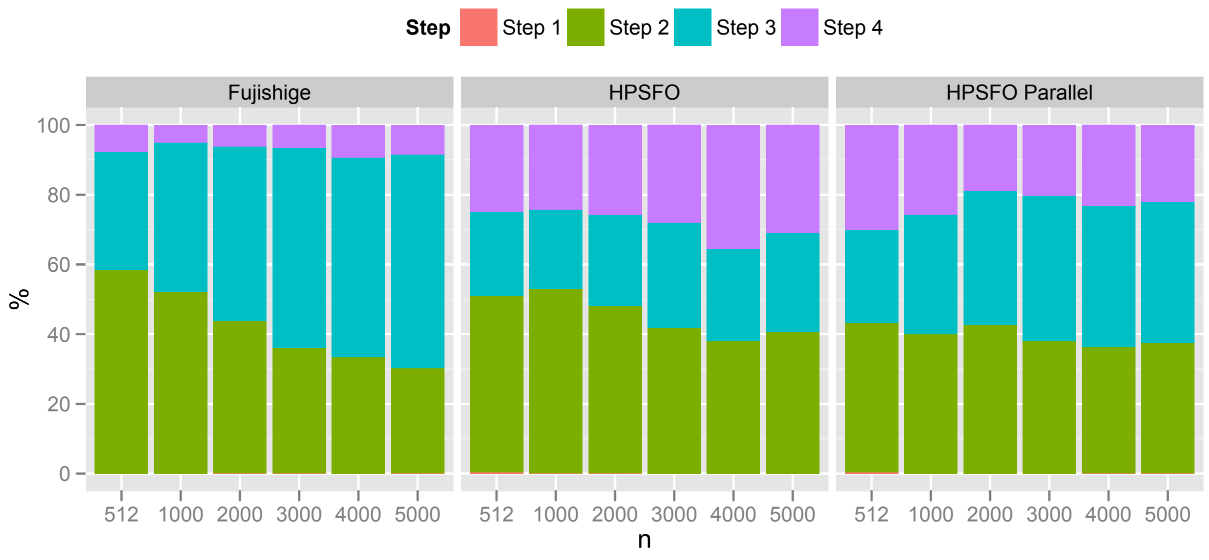


Fig. 6.6: Runtime(s) distribution by Step Min Cut

the broken R matrix as part of Step 4 in parallel using OpenMP as shown in code listing 4.19. Overall, the HPSFO features a better time distribution across steps without any specific Step turning into a clear bottleneck.

The Fig. 6.7 shows the rate of convergence i.e., the Runtime vs the Error and duality gap. We observe how the algorithm quickly drops at the beginning reaching more than 50% of the target Epsilon Error e^{-10} in much less than half the time. Note that we generated this data by tracing the HPSFO implementation using snapshots of 10 iterations, further we have filtered out many points (all those not multiple of 500) to avoid cluttering the plot. The HPSFO uses two convergence criteria, one based on error and the other on the duality gap as explained in section 2. We can see in this Fig. that the criteria that lead to convergence is the Epsilon Error convergence namely the criteria of whether the hyperplane supported by the last point $\hat{x}$ separates the convex hull of S from the origin.

The Fig. 6.8 shows the Histogram where the bins are 10% of the number of columns of the matrices S and R and we count how many columns are deleted at every ten percent of the matrix e.g., we see that as part of the Step 4 of the Wolfe algorithm 1 most column deletions for the Minimum Cut application occur in the first ten percent of the columns i.e., at the be-

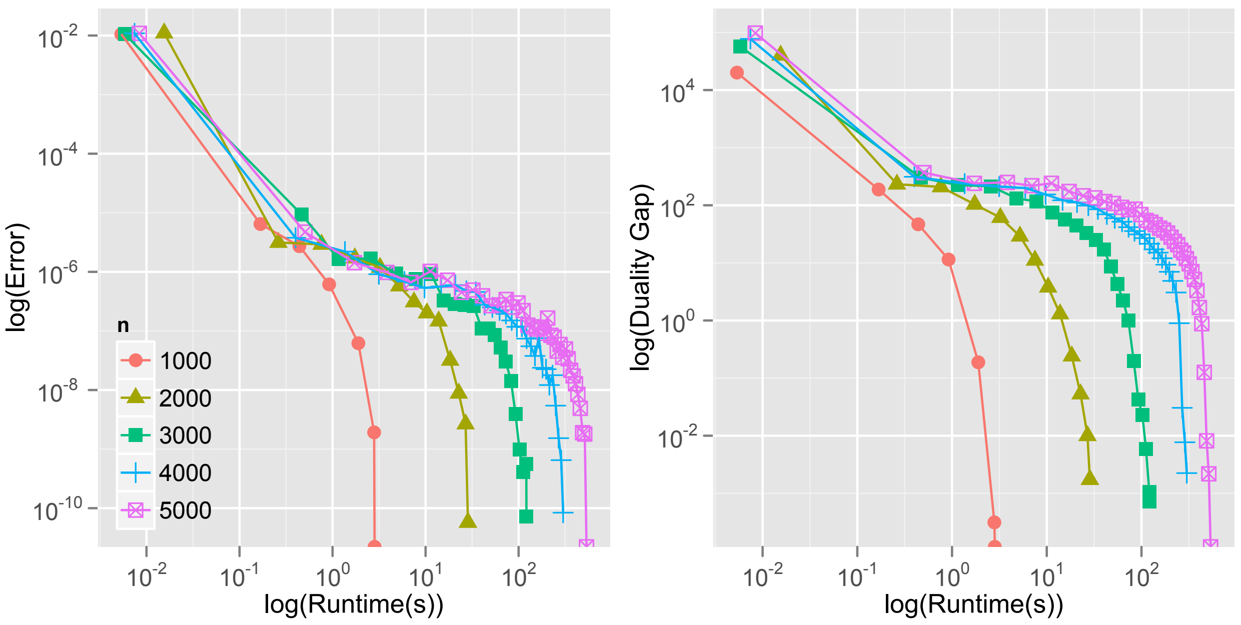


Fig. 6.7: Runtime(s) vs Error and duality gap Min Cut

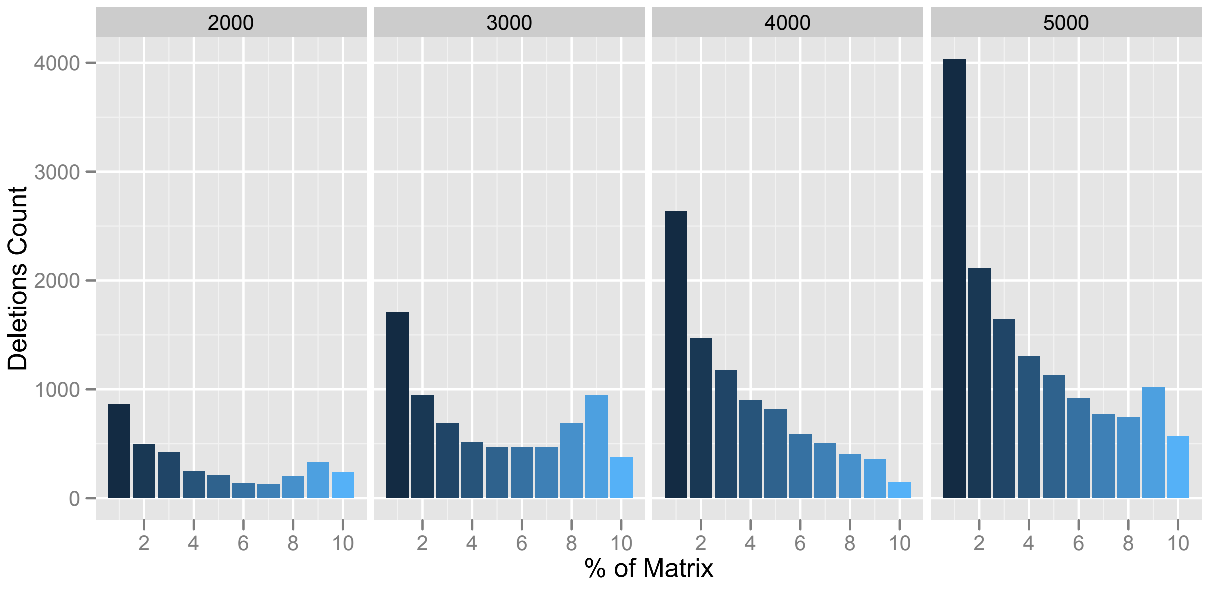


Fig. 6.8: Deletion Histogram Min Cut

ginning of the matrix which is the most costly in every respect. Physically deleting columns at the beginning is very costly and this is what motivated our improvement using the *Gaping* strategy explained in section 4.3.3. The cost of re-triangularizing the matrix R using Givens rotations as explained in section 4.3.4 is also greatly affected by the position of the deleted column j namely delete cost $\approx \frac{(n-j)^2}{2}$ [19]. If j is small or close to zero as in this case, the cost becomes $\approx \frac{n^2}{2}$, in this case the more trailing columns we need to apply the Givens rotations to and the more the application will benefit from higher parallelism and sensible values of the OpenMP Scheduler, more static and bigger chunk sizes. We included this Fig. and discussion to motivate doing *Workload Characterization* [27] and performance analysis per application. Note that the MNP algorithm reveals a consistent behavior with respect to each application, where it tends to make more costly mistakes in some cases. The Minimum Cut application is one example where the MNP Wolfe algorithm tends to make costly mistakes, low j column deletions.

6.3 Log Determinant

In this section we will discuss the experimental results corresponding to the Log Determinant application discussed in section 3.2. We initially tried to write a C++ version of the “two-moons” problem generator included in Bach submodular Package [7] [4] implemented in file `plot_figure_sfm_moons.m`. For this purpose we intended to create our own `tlogdet_moons_generator` C++ version but due to time constraints and also to ensure exact reproducibility of the results comparing to Bach implementation, we decided to instead export the problems generated by the Bach submodular Package implementation and use `tlogdet_moons_generator` to seamlessly import and export. We exported the generated problems as shown in Matlab listing 6.1 using the function `save` including not only the problem input parameters but also the optimal subset, function evaluation and number of iterations as test oracle for our Log Determinant test-suite.

Listing 6.1: Export “two-moons” data from Matlab

```
p  = param_F.p;  % problem size i.e. n=2*p
fv = param_F.FV; % function evaluation on the full set
s  = param_F.s;  % labeled points
K  = param_F.K;  % kernel matrix
```

[7] http://www.di.ens.fr/~fbach/submodular/

```

% run the MNP algorithm
rand ('state',seed);
randn('state',seed);
t = tic;
[x_primal,x_dual,dual_values_mnp,primal_values_mnp,gaps_mnp,
    added1_mnp,added2_mnp,time_mnp,major_iter,minor_iter] =
    minimize_submodular_FW_minnormpoint(F,param_F,100000,1,1e
    -16);
toc(t)

name_txt = sprintf('logdet_two_moons_n%d.txt', p);
name_mat = sprintf('logdet_two_moons_n%d.mat', p);
x_size = size(x_primal, 1);
subopt = primal_values_mnp(size(primal_values_mnp, 1));
save(name_txt, 'p', 'fv', 's', 'K', 'x_size', 'x_primal', '
    subopt', 'major_iter', 'minor_iter', '-ascii', '-double');
save(name_mat, 'p', 'fv', 's', 'K', 'x_size', 'x_primal', '
    subopt', 'major_iter', 'minor_iter');
```

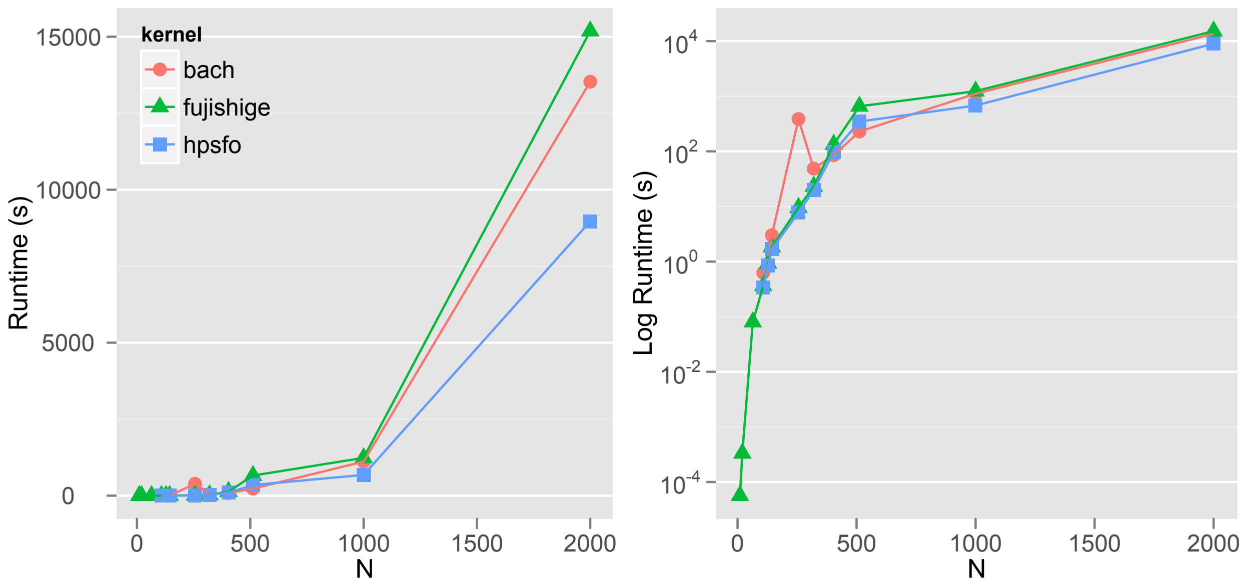


Fig. 6.9: Runtime(s) Log Determinant Non Incremental Evaluation

The Fig. 6.9 compares the execution times for the different SFM kernels. This time we didn't enable EO incremental evaluation to compare one-to-one with Bach MNP implementation. We see that the HPSFO implementation outperforms the others for a small margin and the reason why will be readily apparent after we take a look at the distribution of time per algorithm Step. There are a few important points to discuss regarding this comparison. First, Bach implementation randomizes the starting permutation of the points, again for the sake of fair comparison we changed that to start from the same initial permutation that we used i.e., the sequence $1, \ldots, n$. Second, the Wolfe algorithm 1 (and therefore Fujishige and HPSFO implementations) did not initially converge for the Log Determinant application, we extended the convergence criteria of our Wolfe-based implementations to include and additionally test for the same duality gap convergence as in the Bach submodular Package MNP implementation. Furthermore, even though we could not analyze it in detail due to time constraints, Bach implementation doesn't appear to be robust due to its direct attempt to compute the Cholesky decomposition corresponding to Step 2 of the Wolfe MNP algorithm 1, it would often fail with error

```
not positive definite when adding new point in step 2, exit
```

due to the non semi-positive definiteness of the S matrix after adding a new point. Finally, Bach MNP implementation computes an upper bound on the maximum norm of all points used in the duality gap stopping criteria, and for this, it executes a series of evaluations of the EO function proportional to the size of the ground set as shown in Eq. 2.11, this *pre-scanning* for big problem sizes is very expensive and defeats the purpose. In our HPSFO implementation we kept the duality gap convergence criteria but without computing the ϵ error using this upper bound, instead we use the same input parameter ϵ value (maxed at e^{-10}) as threshold that we used in the initial separating hyperplane convergence test to also test for the duality gap.

The Fig. 6.10 shows the Runtime performance plots comparing the incremental evaluation versions of the MNP kernel implementations: Krause, Fujishige and our favorite HPSFO. Again the results are very tight and we are going to discuss later why this happens. Nevertheless we see a small favorable margin for the HPSFO implementation.

The Fig. 6.11 shows the performance gain due only to incremental evaluation and corresponding to our HPSFO implementation. In the semi-supervised clustering algorithm corresponding to the "two-moons" dataset and evaluating the non-incremental version of the EO function, at every evaluation step we need to compute the Cholesky decomposition of the submatrix corresponding to both the current set X and its complement $X^c = \{V \setminus X\}$,

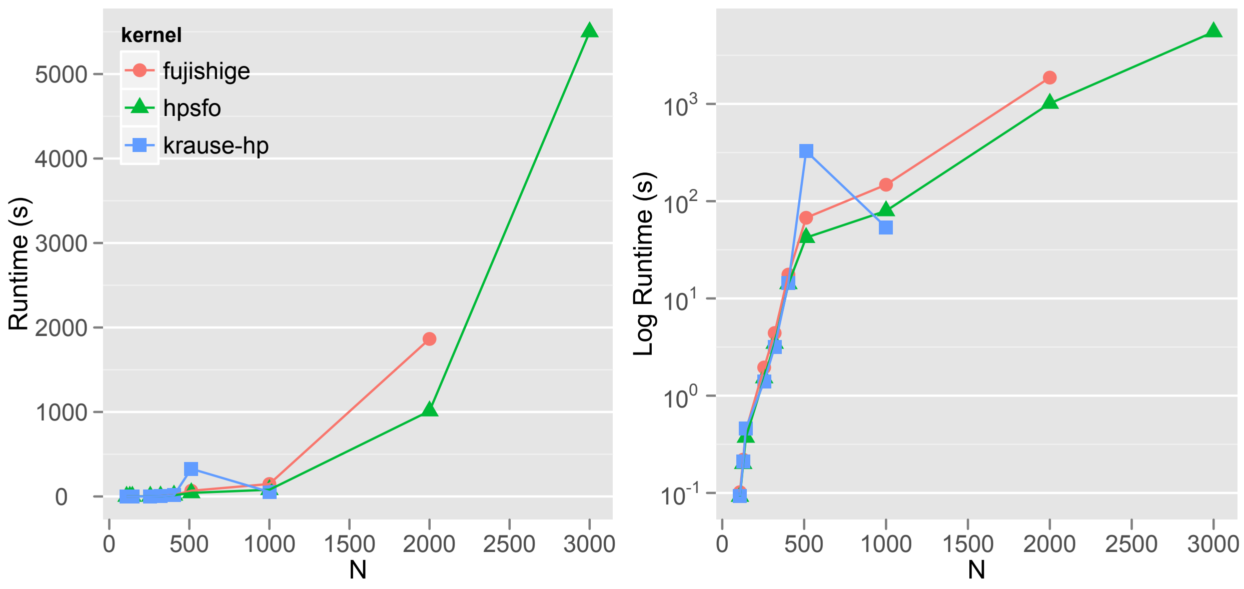


Fig. 6.10: Runtime(s) Log Determinant with Incremental Evaluation

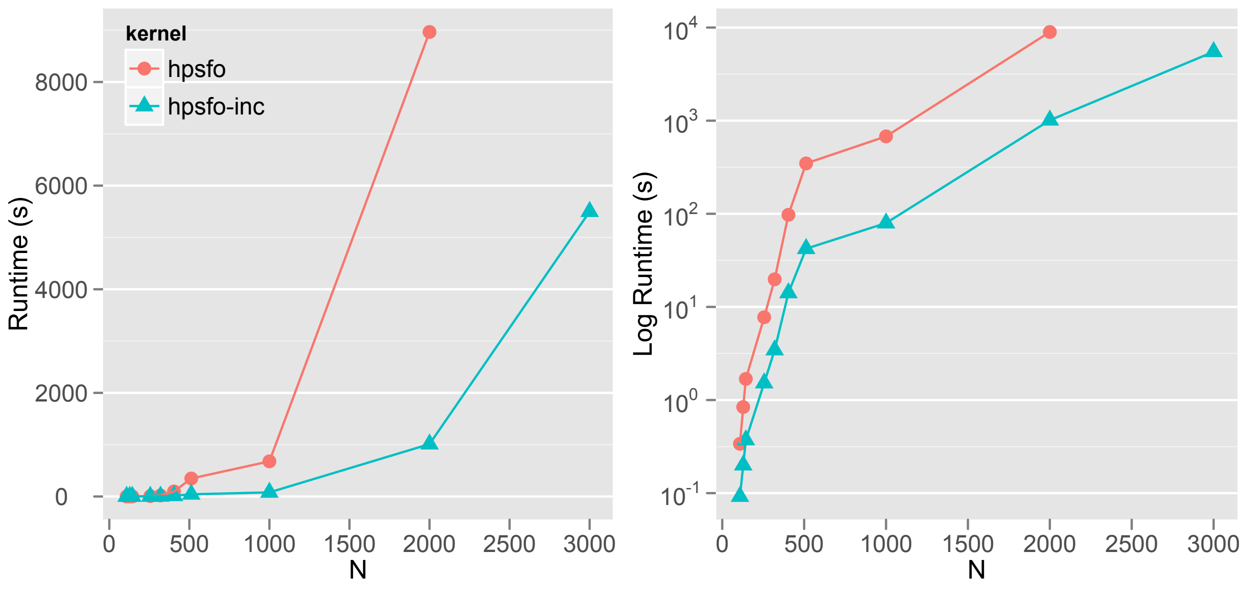


Fig. 6.11: Runtime(s) Log Determinant HPSFO: Incremental vs non Incremental

this cost is roughly $2 \times \frac{n^3}{3}$. By using incremental evaluation we keep in our context the Cholesky decomposition of the two matrices and update one and down-date the other *incrementally* at every EO evaluation. We have discussed the incremental evaluation idea for the Log Determinant application in section 3.2. The cost in flops for up-dating and down-dating a Cholesky decomposition are $(2mn + 4n + 3)$ and $\left(\frac{(n-j)^2}{2}\right)$, respectively. In this case we have successfully reduced the cost and complexity of the EO evaluation from $O(n^3)$ to $O(n^2)$ and this is reflected in our Runtime Fig. 6.11 where we can observe a gain of roughly one order of magnitude in runtime. It is important to note that the incremental EO function evaluation corresponding to the Log Determinant application heavily uses our high-performance foundation 4.3.1 e.g., fast matrix mutations 4.3.3 and fast orthogonal updates 4.3.4.

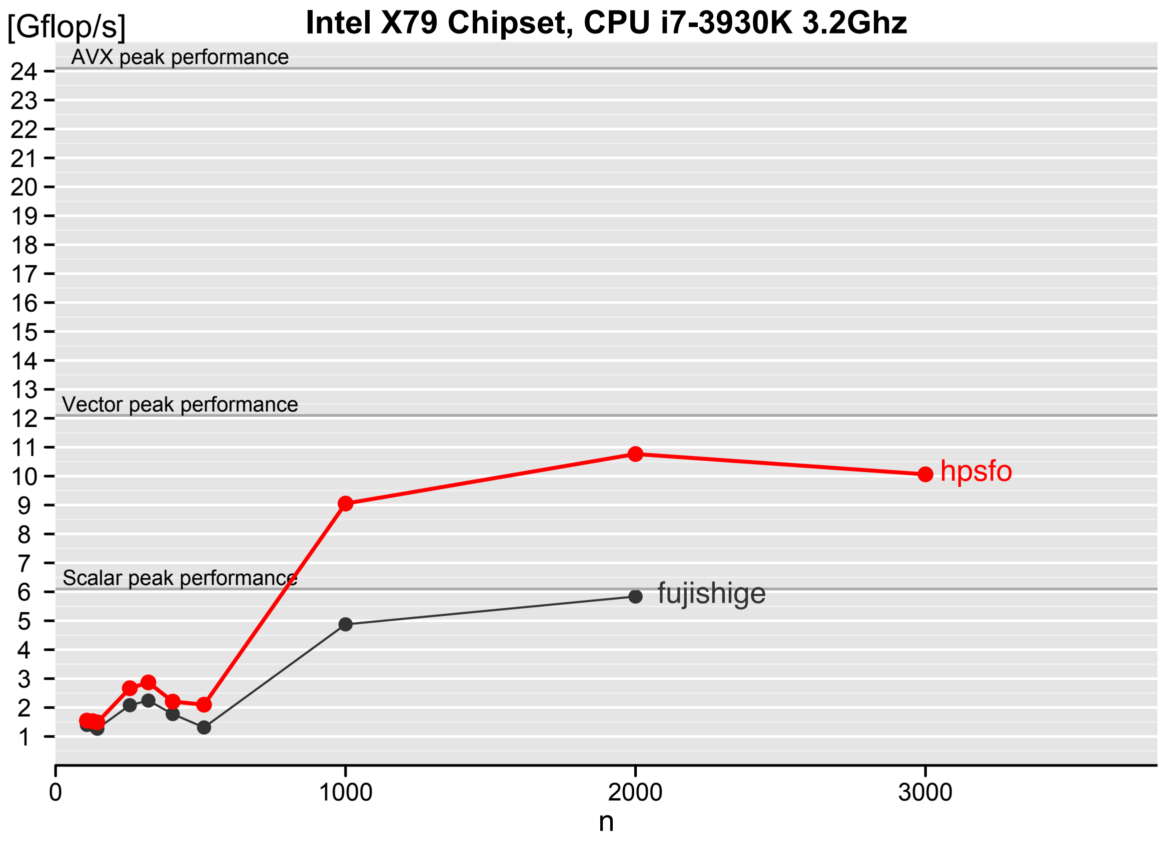


Fig. 6.12: Gflop/s Log Determinant with inc. evaluation HPSFO vs Fujishige

The Fig. 6.12 shows the Gflop/s performance plot comparing Fujishige vs HPSFO. In this case we withdrew some of our high-performance improvements applied to the EO incremental evaluation for the Fujishige version, namely we withdrew the fast matrix mutations 4.3.3 and so it uses the column deletion without our *Gaping* strategy optimization. In this specific application and in the incremental EO function evaluation case we put under stress test our high-performance foundation, in particular, our fast matrix mutations 4.3.3 and, more specifically, the fast Givens rotations using Register blocking and loop unrolling 4.3.4, since this is the exact same procedure [8] required to down-date an existing Cholesky decomposition [48] we successfully "killed two birds with one *fast* bullet". We can happily observe that continuously executing our fast Givens implementation contributes to getting up to 90% of SSE2 vector peak performance.

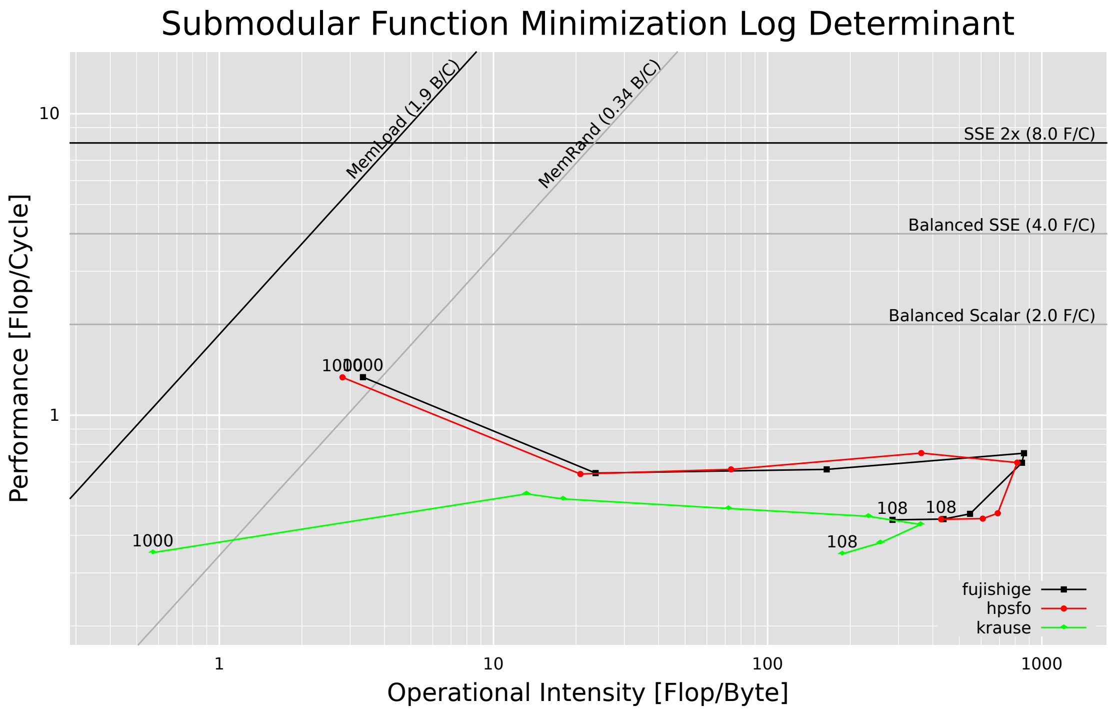


Fig. 6.13: Roofline Log Determinant with inc. evaluation

The Fig. 6.13 shows the Roofline comparing Fujishige vs HPSFO and in the Intel Core Duo platform 6.2. Here they both have the exact same high-performance EO evaluation implementation. We can see they perform very

[8] While the procedure is the same, we need a more specialized Givens rotation generator function that will produce non-negative matrix diagonal elements, i.e., `dlartgp`

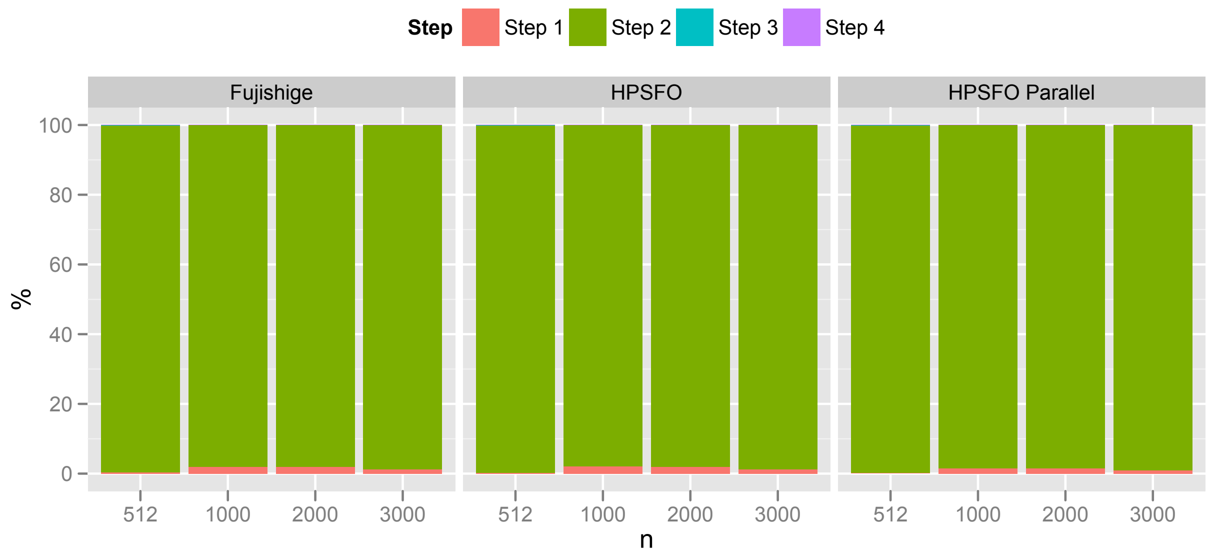


Fig. 6.14: Runtime(s) distribution by Step Log Determinant

closely. The reason why there are no big differences between the different MNP algorithms and for the Log Determinant becomes readily apparent in Fig 6.14. Here we see that 99.9% of the time is spent doing function evaluation. The number of iterations is also very low as to make any big difference between the algorithms: 73, 76 and 123 major iterations for problem sizes 1k, 2k and 3k respectively; in this application the MNP algorithm also makes very few "mistakes" and this we can see also in Fig 6.16. It becomes clear that for this application the highest performance pay-off would go into optimizing the EO evaluation function which we already did by implementing the incremental evaluation version.

The Fig. 6.15 shows the convergence rate of the Log Determinant application for the Error as well as the duality gap. We can see that convergence is driven by the duality gap criteria and that the duality gap decreases slowly until it drops sharply. Comparing to Bach's original problem setup and implementation, we have the exact same number of iterations. Furthermore, this plot reveals an interesting detail and possible improvement over Bach implementation that we took advantage of. Bach implementation and for the purpose of duality gap convergence test, pre-computes an Epsilon that takes the maximum norm which is built by evaluating all singletons using the EO evaluation, this is very expensive, say for sizes over 10k and not really needed

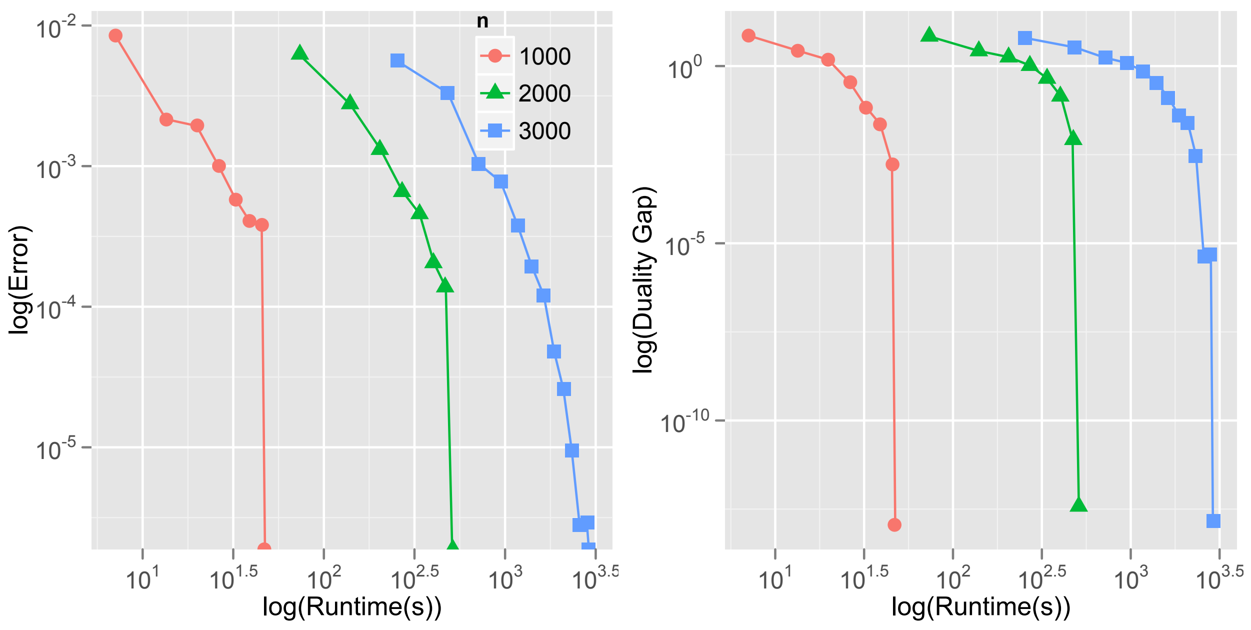


Fig. 6.15: Runtime(s) vs Error and duality gap Log Determinant

since the duality gap convergence as we can see in these plots decreases slowly until it makes a sharp drop. Therefore, we can simply make the duality gap Epsilon configurable rather than pre-computing it exactly.

The Fig. 6.16 shows the column deletion histogram corresponding to the Log Determinant application. It is clear that for this case the Wolfe MNP algorithm doesn't make too many mistakes but rather a few and all costly i.e., point deletions as part of the Step 4 of the algorithm occur always at the beginning of the matrix or first 10% of the columns of the matrices S and R.

After reviewing Fig. 6.14, it reveals the fact that all the time is spent doing incremental EO function evaluation, which in turn means we spend most of the time running our fast Givens rotations Register blocking and loop unrolling implementation. Therefore, we enabled parallelism at both MKL and OpenMP and configure OpenMP to the best configuration found `export OMP_SCHEDULE=static,70` [9]. The Fig. 6.17 shows the near $2\times$ parallel speed up we gain for this specific problem and it is mainly thanks to our fast orthogonal update implementation.

[9]We would need experimental design and ANOVA analysis in addition to workload characterization to get the best factor levels for each application.

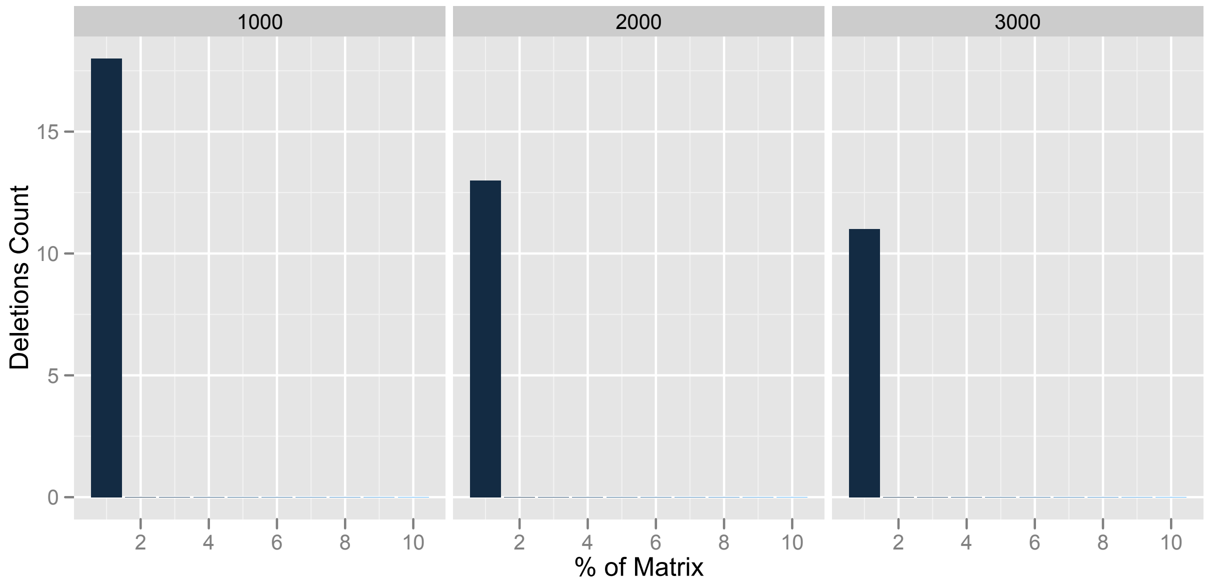


Fig. 6.16: Deletion Histogram Log Determinant

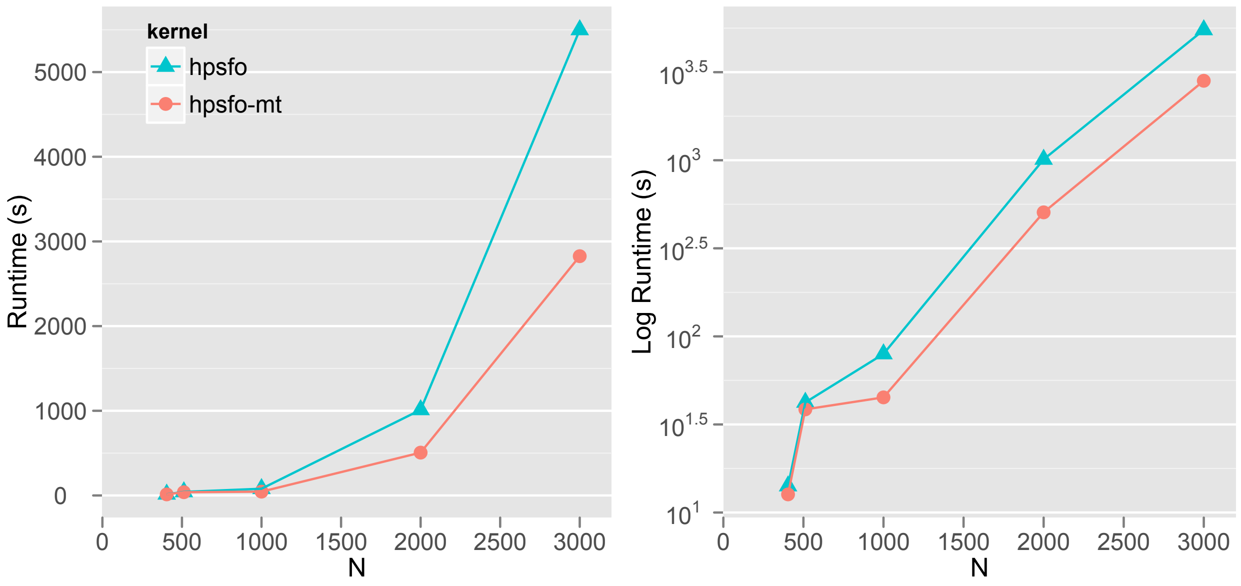


Fig. 6.17: Runtime(s) Log Determinant HPSFO OpenMP static,70

6.4 Corpus Selection

We obtained the data for the Corpus Selection from the authors of the publication "Optimal Selection of Limited Vocabulary Speech" [33]. Their datasets were uniform meaning the amount of information associated with the optimal set of utterance was the cardinality of the utterances set. The resulting submodular function did not behave very well since we would get a flattened function image and quickly find any result with low number of iterations. We fixed that by associating a weight to each utterance i.e. a weight between the Source node and each Utterance node, we choose the weights values to be very skewed. By doing so we effectively increased the selectivity of the optimal subset and the MNP algorithm then produced more interesting results.

The Fig. 6.18 shows the comparison between Fujishige and HPSFO kernel implementations and using the EO incremental function evaluation in both and for the Corpus Selection application. We observe that our HPSFO implementation features above 10× speed up over Fujishige base.

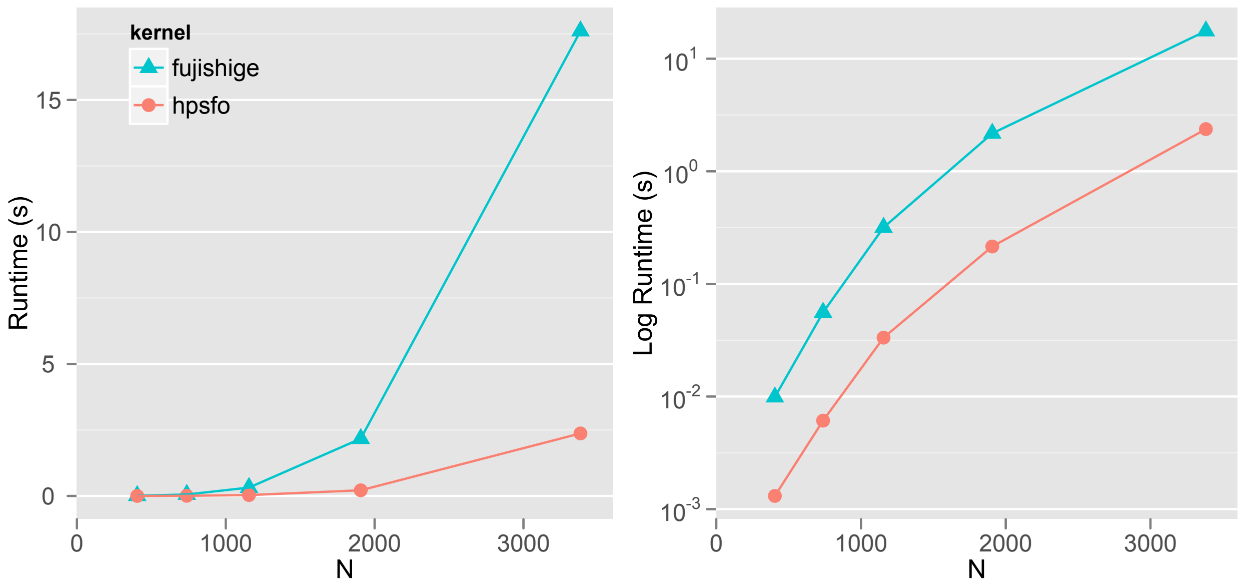


Fig. 6.18: Runtime(s) Corpus Selection with Incremental Evaluation

The Fig. 6.19 shows the net gain from using non-incremental EO function evaluation to the incremental version. We successfully reduced the complexity of the EO function evaluation from $O(\mathrm{n}^2 \log n)$ to $O(n)$.

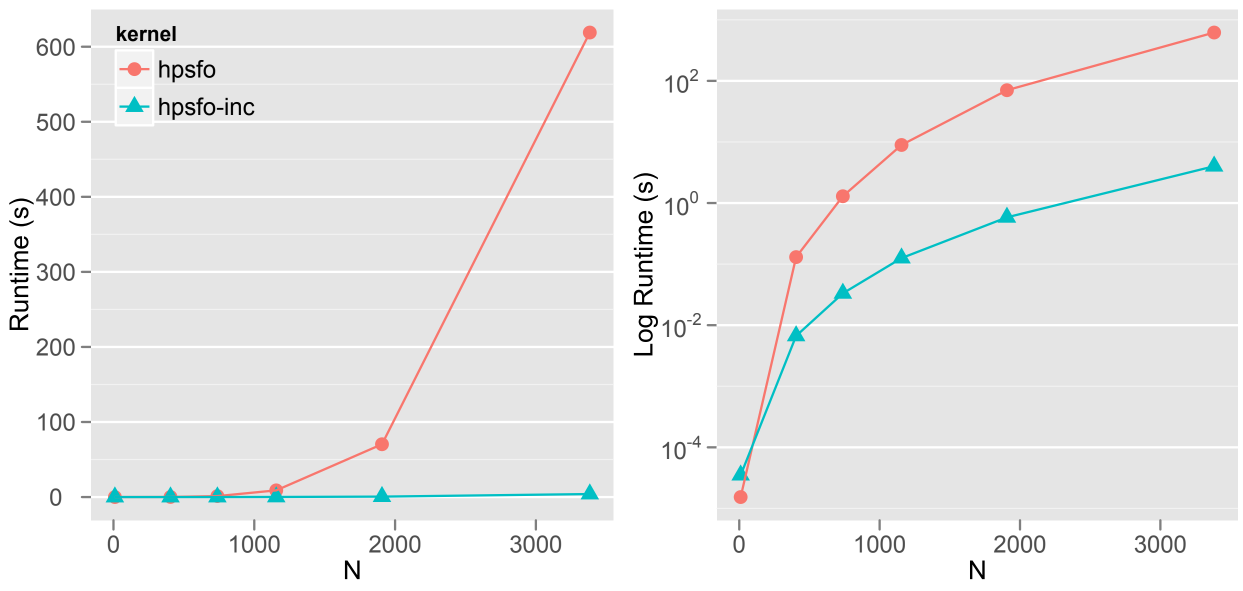


Fig. 6.19: Runtime(s) Corpus Selection HPSFO: Incremental vs non Incremental

The Fig. 6.20 shows the Gflop/s plot once more comparing the two main contenders and we see how our HPSFO implementation not only outperforms Fujishige-Wolfe but also scales relatively well with respect to the problem sizes. The problem sizes we obtained for the Corpus Selection [33] were not really big and an optimal is found after only a few seconds.

The Fig. 6.21 shows the Roofline plot resulting for the Corpus Selection. This confirms the results we obtained in our standard performance Gflop/s plot, here we also see higher performance measured in billion flop operations per second.

The Fig. 6.22 shows the distribution of the Runtime for the Corpus Selection for every Step of the Wolfe algorithm 1. Note how the time spent doing Step 3 is non-negligible, and as seen before this correlates with the higher performance we observed in the previous performance plots that favor our HPSFO implementation. Furthermore, note how the Step 3 again becomes the performance bottleneck for Fujishige implementation. The Step 3 of Wolfe algorithm finds the MNP within the affine hull **aff** S and does so by solving the systems previously shown in Eq. 2.17.

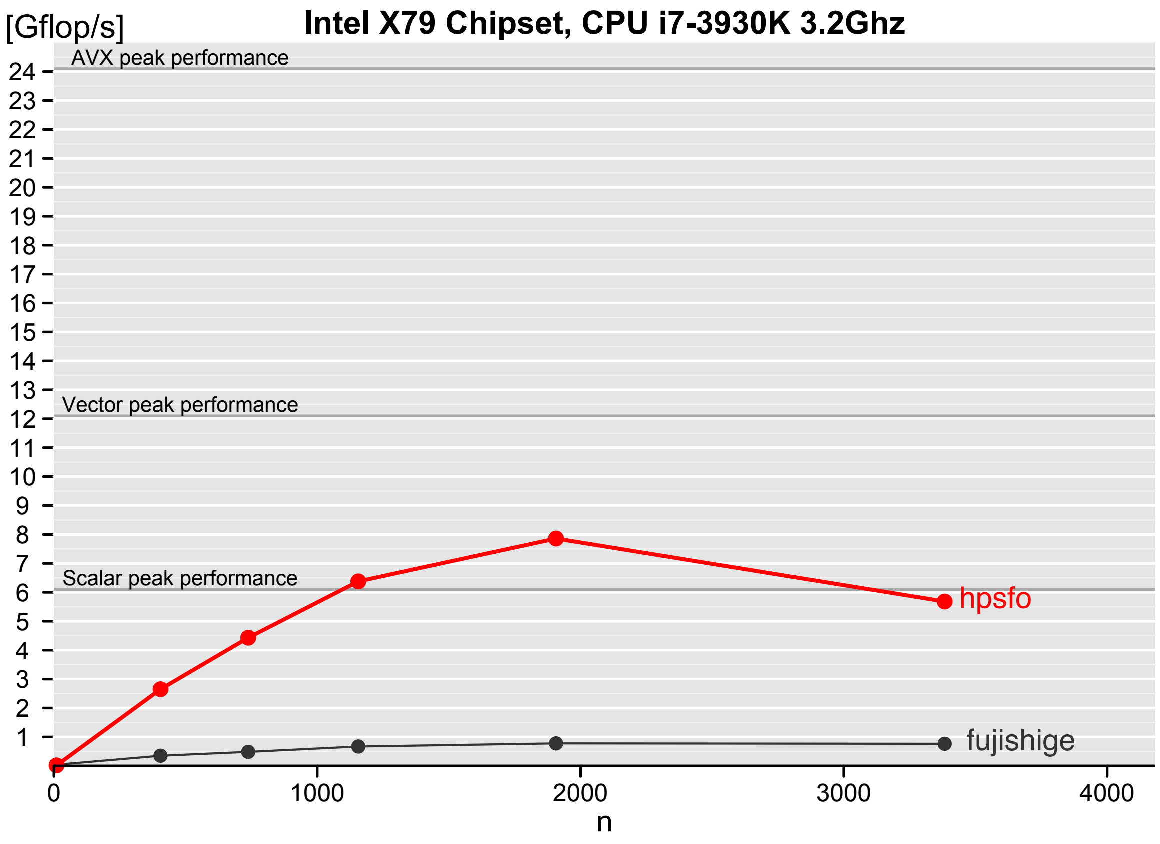


Fig. 6.20: Gflop/s Corpus Selection with inc. evaluation HPSFO vs Fujishige

In Fig. 6.23 we can see the details of the convergence for the Corpus Selection application. Again, we have very nice fast initial drop of the Error and convergence led by the duality gap criteria.

6.5 Fastest showcase

In this section we will only emphasize how better our final fastest HPSFO is performance-wise, by first breaking the number of iterations compatibility with the Fujishige-Wolfe base [10] implementation using a more efficient sort-

[10]In all our previous benchmarks we utilized the same `quicksort2` implementation by Fujishige not to break the compatibility with the improved HPSFO with respect to the number of algorithm iterations.

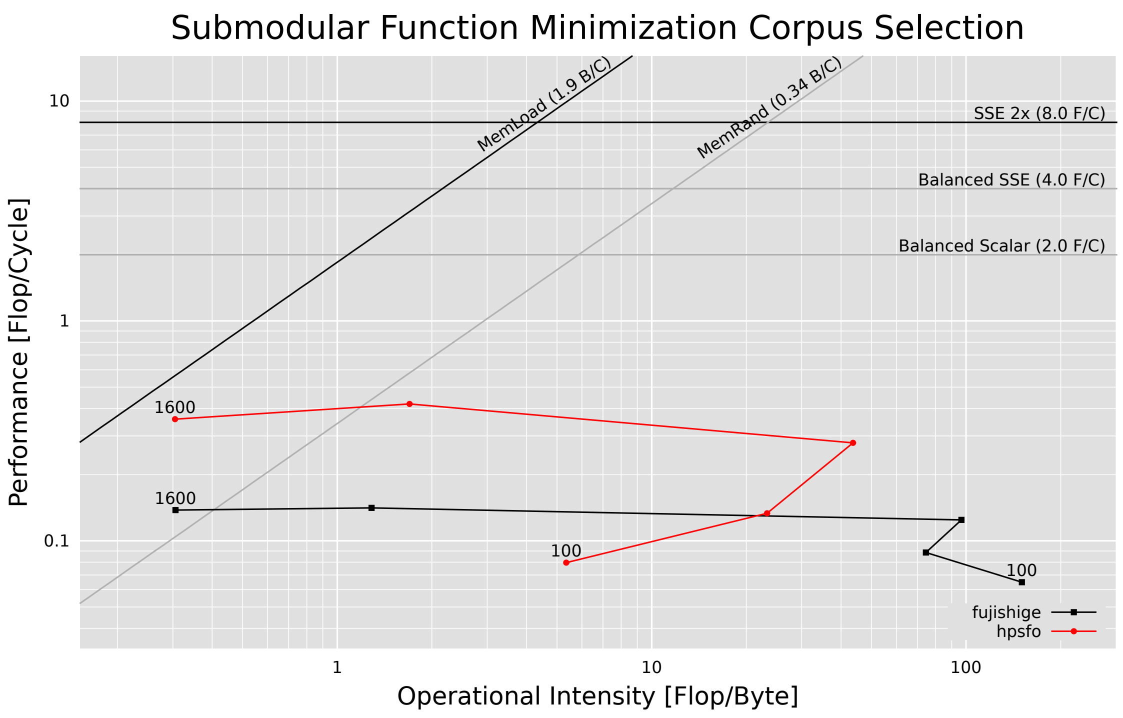


Fig. 6.21: Roofline Corpus Selection with inc. evaluation

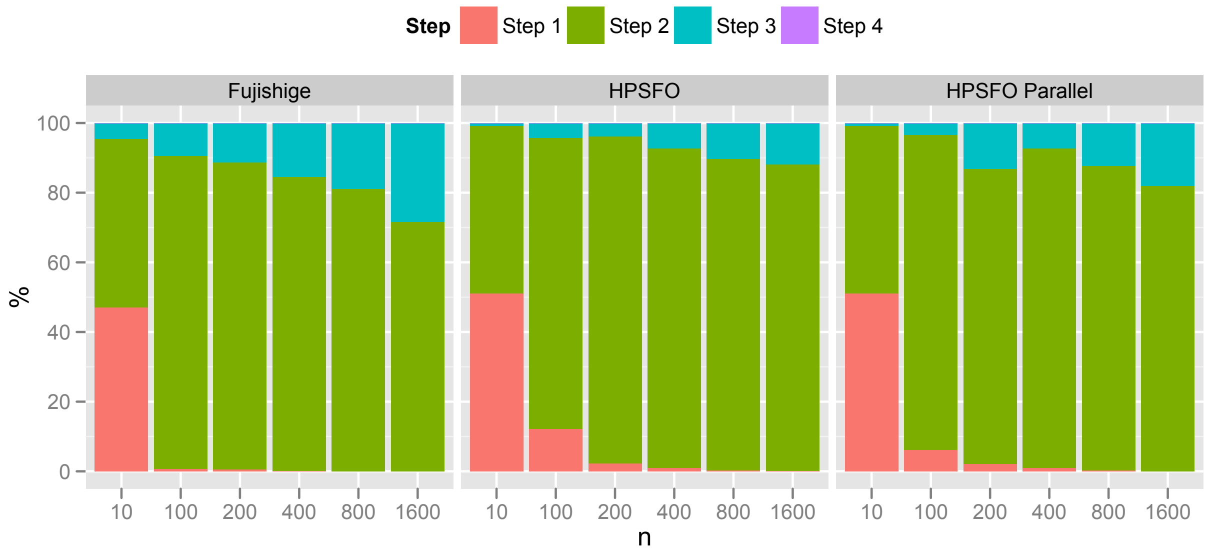


Fig. 6.22: Runtime(s) distribution by Step Corpus Selection

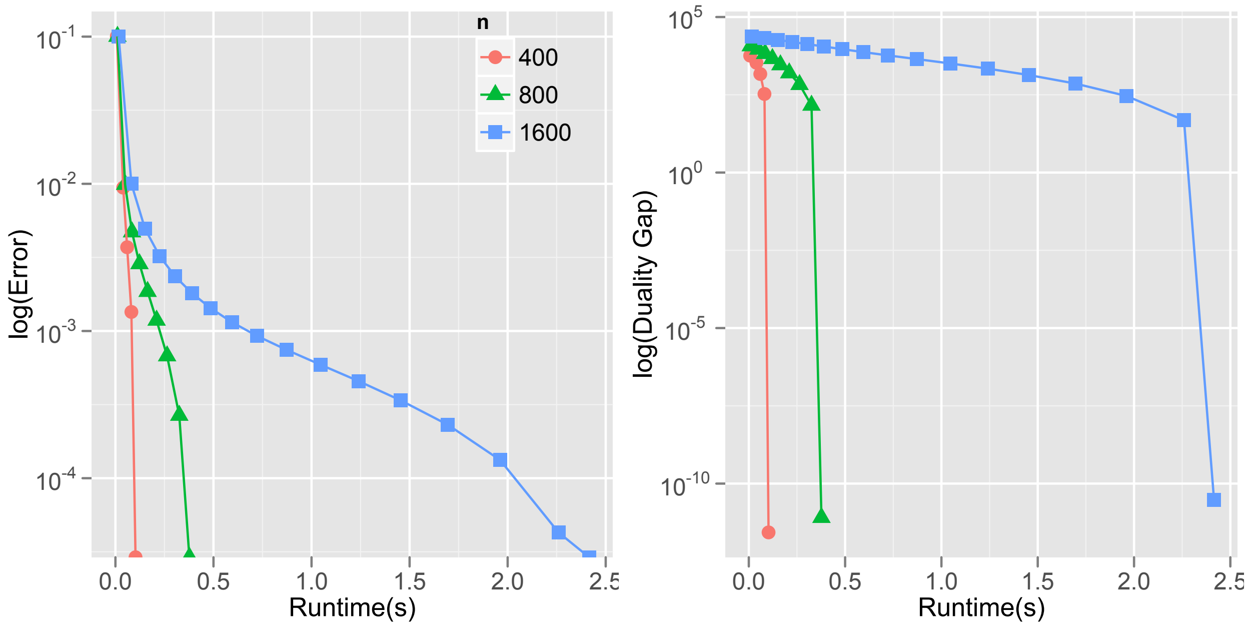


Fig. 6.23: Runtime(s) vs Error and duality gap Corpus Selection

ing algorithm `std::sort` or TBB[11] `tbb:parallel_sort` as part of the Greedy Edmonds algorithm step 2 and 3. The reason why changing the sorting algorithm leads to different number of iterations is due to how the sorting algorithm stirs elements that have the same value, effectively leading the algorithm to explore different points in a different order. This behavior is conceptually the same to providing an initial random permutation of the indexes, a feature available in our framework implementation that we could have also explored and is left for future work.

The Fig. 1.1, 1.2 and 1.3 show the overall gain of our final and fastest parallel HPSFO implementation compared to Fujishige-Wolfe. The performance gap increases with the problem sizes for all applications. For small problem sizes we disabled parellelism and only enabled it for sizes where it would make a difference.

The Fig. 6.24 shows the convergence rate of our HPSFO implementation for a 10k Minimum Cut Wide problem. In this case we are looking at 17'393 major and 24'546 minor iterations for a total of 8.199358e+12 flops. Here we see again a sharp drop of the Error within the first few seconds and a slower Error-led convergence thereafter. We can either use a very fast few-seconds

[11]`http://http://threadingbuildingblocks.org/`

solution at $\epsilon = e^{-6}$ Epsilon-Error or a high-quality no-compromise solution at $\epsilon = e^{-10}$ Epsilon-Error that would otherwise take 25 minutes.

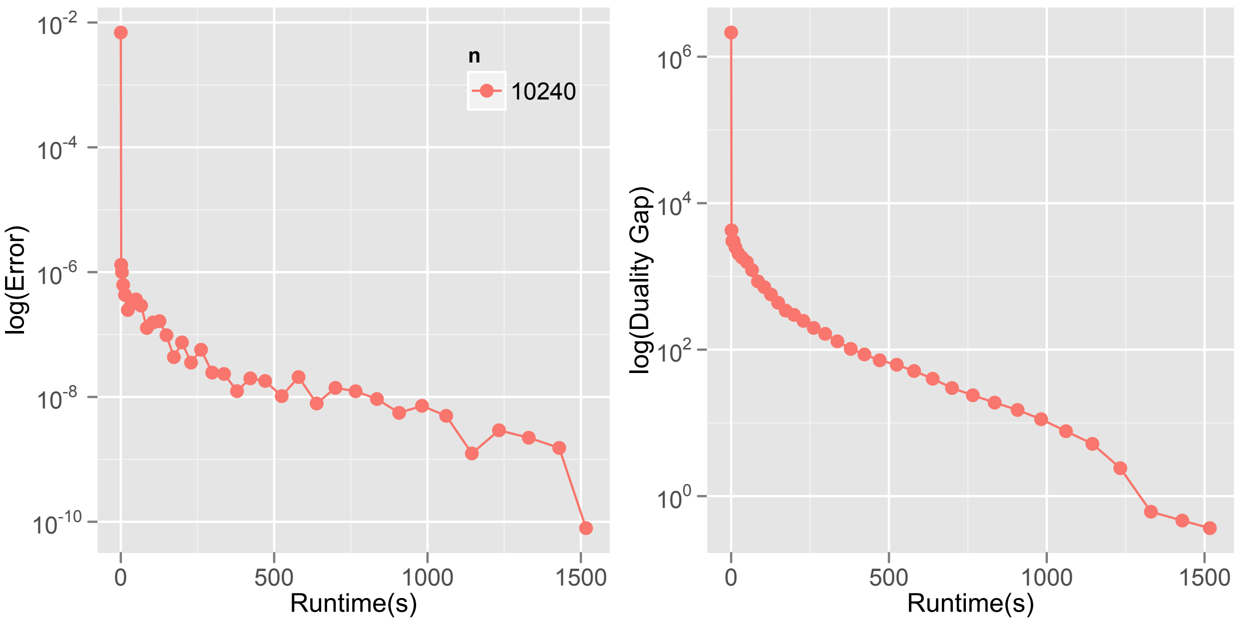


Fig. 6.24: Runtime(s) vs Error and duality gap Min Cut

The Fig. 6.25 shows the convergence rate of our HPSFO implementation for a 10k Log Determinant problem (over 2.3GB data file). In this case we are looking at 109 major and 111 minor iterations for a total of 6.708646e+15 flops. Here we see a very slow drop of the Error and duality gap that takes well over 22 hours and ends finally in duality gap-led convergence.

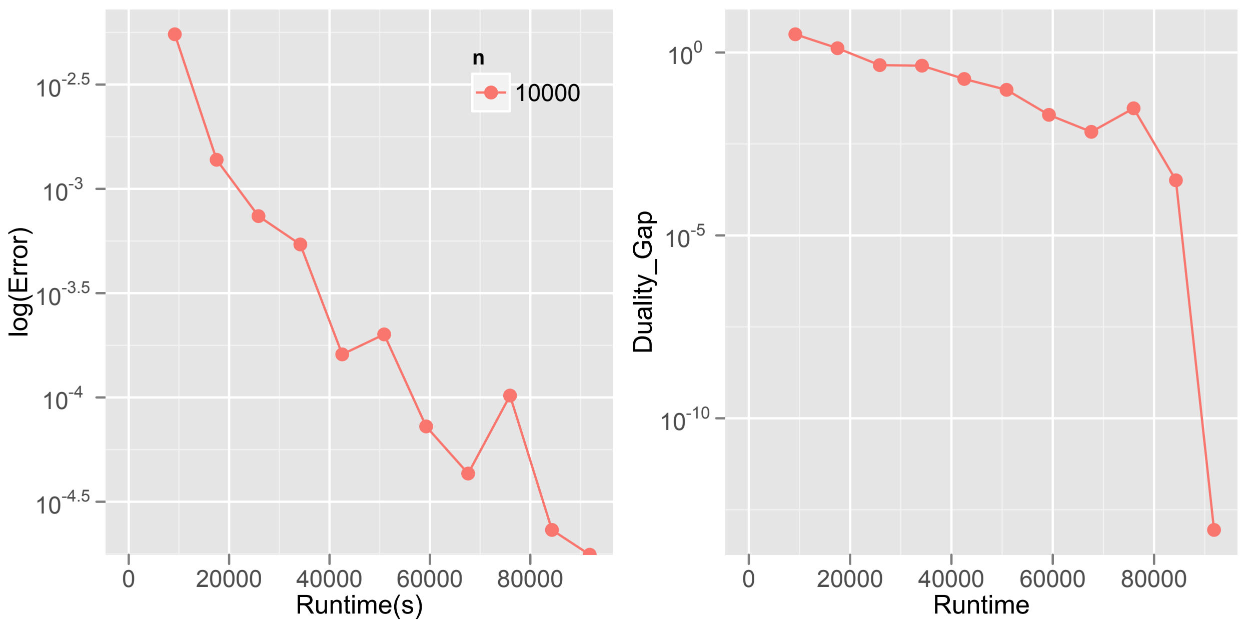


Fig. 6.25: Runtime(s) vs Error and duality gap Log Determinant

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this work we have optimized two existing MNP kernels for general SFM: Krause [30] and Fujishige-Wolfe [13, 49]. We have successfully cast their implementation in terms of our high-performance foundation that builds on top of Intel MKL and Intel TBB. By building on top of Intel MKL and Intel TBB, our implementations will automatically benefit from continuous updates and improvements to those libraries, e.g., support for more advanced vector extensions AXV2 [1]. We successfully optimized Krause implementation to take advantage of fast orthogonal updates and we did so by "sandwiching" different QR update types. In this case we reduced the cost and complexity of its generic kernel implementation delivering a speed up of up to 12× times faster over Krause Matlab Toolbox MNP implementation. We reused the same high-performance foundation built in the abstractions `tsfo_vector<T>`, `tsfo_matrix<T>` and `tsfo_matrix_tria<T>` to optimize Fujishige-Wolfe MNP algorithm. Instrumenting both MNP implementations and validating our findings using Performance counters with PAPI we realized that Fujishige-Wolfe implementation was algebraically much simpler and offered lower cost, we therefore took Fujishige-Wolfe as base to build the final and fastest HPSFO implementation. Our final HPSFO implementation successfully outperforms all others and for all the applications we tested. Note that in some cases, e.g., the Log Determinant application, the Fujishige MNP im-

[1] `http://software.intel.com/en-us/articles/haswell-support-in-intel-mkl/`

plementation would indirectly benefit from our high-performance foundation via the function evaluation EO. Moreover, Fujishige-Wolfe implementation only benefits from compiler improvements, whereas our HPSFO directly benefits from compiler and improvements to Intel high-performance libraries, i.e., BLAS, LAPACK. One important result of the performance analysis of the MNP algorithm is that it is fundamentally memory-bound, as the Roofline plot Figures revealed, e.g., 6.5 and as we have discussed this is rooted in the low reuse nature of the algorithm, therefore the MNP would directly benefit from higher memory transfer rates. In conclusion, we have built a high-performance HPSFO implementation that features both types of parallelism SIMD and MIMD.

In addition to the generic high-performance optimizations applied to the generic Krause and Fujishige-Wolfe MNP kernels, we also implemented the algorithmic complexity improvements originally described by Krause as *incremental update* evaluation of the EO function and as part of the Greedy step. We successfully lowered the complexity of the EO function evaluation for all applications and therefore their runtime.

Finally, we built a C++ software framework that allows coherent coexistence of different SFO algorithms and applications. Furthermore, our framework is designed for extensibility, it provides a "zero-dependency" API that allows reuse from different platforms, e.g., Java JNI and offers built-in generic integration with Matlab. We have successfully delivered a high-performance platform for SFM, with a very fast implementation HPSFO that is cross-platform portable, easy to reuse, extend and maintain.

7.2 Future work

There are several areas with opportunities for further research and improvements. First, it would be a welcome addition to further utilize TBB within our high-performance foundation to exploit extra parallelism for possible bottleneck loops where auto-vectorization and OpenMP are not applicable. Auto-vectorization or for that matter intrinsics are, in general, not applicable for loops that access non-contiguous memory locations. Similarly, OpenMP is not applicable for loops which are not in canonical form [2].

In section 4.3.4 we described our fast Register blocking with loop unrolling implementation to repair a previously upper triangular matrix after a column deletion due to "mistakes" made by the MNP algorithm. It is possible that the algorithm make mistakes where multiple columns need to be deleted, so far our workload applications were not affected by this case, it is rather rare but it happens. We implemented a general re-triangularization using Givens rotations that would work for deleting any number of columns and not just one but it didn't outperform doing several runs of a single update sweep. Note that a general implementation for any number of columns can not be implemented with Register blocking and loop unrolling since the number of deleted columns needs to be fixed to develop such function. It would be an interesting approach left for future work to re-triangularize a matrix in upper Hessenberg form when more than one column is deleted and do so faster than our blocked Givens approach, or perhaps build a code generator that will create Register blocking and loop unrolling functions for different number of deleted columns. Furthermore, it would be a welcome addition to implement and test additional block sizes even if done manually as discussed in section 4.4.

We have provided very high test coverage, production-level testing indeed and for all the applications we implemented. In the case of the Minimum Cut we were able to test and cross check the results between Krause and Fujishige-Wolfe [13] implementations making sure the outputs where correct in every case. We then generated a test-oracle and implemented regression test-suites to make sure that future delta improvements would not break the correctness of the algorithm implementations. We even check the number of iterations to make sure that future performance improvement changes leave the algorithm functional behavior invariant. We did so too for the Log Determinant application where we succeeded to match the results of Bach implementation [4] exactly. It would be nevertheless, a great improvement to provide performance regression test-suites to check that delta improvements do not wind up degrading performance. Having such performance test-suites plus a diligent tracking of performance results via a version control system would ensure that there is no unexpected performance degradation due to seemingly innocuous code changes.

An important but trivial addition left for further work, would be to create a additional Matlab adapter corresponding to the incremental update of the EO evaluation function. In this case we simply require introducing a new class implementation, i.e., `tmatlab_sf_inc_adapter` that subclasses both `tmatlab_sf_adapter` and `tabstract_sf_inc_context` and add this incremental variant as part of the "zero-dependency" Façade implementation `sfo_matlab_adapter.cc`. Doing so will allow future Matlab applications to take advantage of the more efficient incremental evaluation EO.

As discussed previously, we have integrated the Iwata SFM kernel `tiwata_sfm_scaling_kernel` as part of our framework but due to mistakes in either side we could not get it to pass all of our application test-suites. Due to time constraints we could not fix this and it would be a great improvement to get the Iwata SFM kernel to pass all test-suites and then get performance results using our benchmark framework implementation `benchmark.cc`.

Next, the runtime of the high-performance MNP kernels is affected by many factors and levels:

- Environment variable `MKL_NUM_THREADS`: number of Threads available to Intel MKL has levels 1,…,c where c could be the number of cores available to the underlying platform or even more.

- Environment variable `OMP_NUM_THREADS`: number of Threads available to OpenMP has levels 1,…,c where c could be the number of cores available to the underlying platform or even more.

- Environment variable `OMP_SCHEDULE`: that contains scheduler type (`static` , `dynamic`, `guided`, `auto`) and chunk size which is an integer number and defines the block size assigned to one thread, , e.g., `static,70`. The Register blocking with loop unrolling solution that computes fast Givens rotations to update the R matrix has the innermost loop where OpenMP is used as shown in code listing 4.19 is set to be configured from environment via this `OMP_SCHEDULE` environment variable.

- Triangularization `NB` blocking size. So far we have implemented blocking sizes 0,2,4,7,8,15 and 16. In general bigger block sizes tend to perform best for larger problem sizes and take more advantage of parallelism via OpenMP whereas smaller sizes, e.g., NB=4 are best suited for single threaded and fast execution of smaller problem sizes. A batch process that would require computing many problems of similar characteristics, e.g., size etc would benefit from fine tuning this NB size to that specific workload.

- Triangularization “mode”: so far we have three triangularization implementations `blas3`, `openmp` and `auto-vectorization` and their results may vary in the future, e.g., once AVX2 is available, or Intel MKL provides an inplace implementation of MMM of the form $A = \alpha \cdot AB$

- Givens rotation primitive: we have `sqrt-based`, `cblas_drotg`, `dlartg` and `dlartgp`. Some problems may take advantage of using the faster but less accurate `cblas_drotg` implementation.

Some of these factors would naturally interact leading in some cases to unexpected performance results, e.g., the `NB` triangularization block size and `OMP_NUM_THREADS` would clearly interact. A further work where we do workload characterization depending on the application would also bring an extra advantage. If we know where most of the time for that application is spent and how, by exploring the tracing results we saw, e.g., in 6.6 and 6.8 then we can better tune the different parameters accordingly. One can imagine that if an application makes a lot of mistakes or Step 4 of the Wolfe algorithm 1 and the mistakes are more costly, i.e., the deleted columns are at the beginning of the matrices, then the algorithm will perform best for a given triangularization mode and `NB` size and a specific `OMP_SCHEDULE` setting. Therefore we would propose as extension of this work to first externalize all these factors to, e.g., environment variables. Then develop an Experimental design, e.g., 2^k factorial design and ANOVA to study the effects of different parameter combinations in the runtime and pick the best for the specific application at hand.

Bibliography

[1] E. Anderson, J. Dongarra, and S. Ostrouchov. Lapack working note 41: Installation guide for lapack. Technical report, Knoxville, TN, USA, 1992.

[2] P. Arbenz. Parallel numerical computing eth zurich, 2012.

[3] F. Bach. Convex Analysis and Optimization with Submodular Functions: a Tutorial.

[4] F. Bach. Learning with submodular functions: A convex optimization perspective. *CoRR*, abs/1111.6453, 2011.

[5] T. Badics. Maxflow generator in dimacs format, 1991.

[6] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, New York, NY, USA, 2004.

[7] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Addison-Wesley Publishing Company, USA, 2010.

[8] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled qr factorization for multicore architectures. In *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, PPAM'07, pages 639–648, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] S. Chellappa, F. Franchetti, and M. Püschel. How to write fast numerical code: A small introduction. In *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 5235 of *Lecture Notes in Computer Science*, pages 196–259. Springer, 2008.

[10] I. Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Number 248966-018. March 2009.

[11] J. Edmonds. Combinatorial optimization - eureka, you shrink! In M. Jünger, G. Reinelt, and G. Rinaldi, editors, *Combinatorial optimization - Eureka, you shrink!*, chapter Submodular functions, matroids, and certain polyhedra, pages 11–26. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[12] S. Fujishige. *Submodular Functions and Optimization (2nd ed.)*. Elsevier Press, Amsterdam, The Netherlands, 2005.

[13] S. Fujishige, T. Hayashi, and S. Isotani. The minimum-norm-point algorithm applied to submodular function minimization and linear programming. 2006.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, Nov. 1994.

[15] Z. C. Giovanni Azua, Andrei Fruza. How to write fast numerical code project: Submodular function optimization for the s-t graph cut, 2010.

[16] G. H. Golub and M. A. Saunders. Linear least squares and quadratic programming. Technical report, Stanford, CA, USA, 1969.

[17] G. H. Golub and C. F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[18] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[19] S. Hammarling and C. Lucas. Updating the qr factorization and the least squares problem, 2008.

[20] Intel. Core-i7 lga 2011 datasheet volume-1, 2011.

[21] Intel. A guide to vectorization with intel ®c++ compilers, 2012.

[22] Intel. Intel compiler user and reference guide, 2012.

[23] Intel. Intel math kernel library, 2012.

[24] S. Iwata. A faster scaling algorithm for minimizing submodular functions. *SIAM Journal on Computing*, 32:833–840, 2001.

[25] S. Iwata, L. Fleischer, and S. Fujishige. A combinatorial strongly polynomial algorithm for minimizing submodular functions. *J. ACM*, 48:761–777, July 2001.

[26] S. Iwata and J. B. Orlin. A simple combinatorial algorithm for submodular function minimization. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 1230–1237, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[27] R. Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling.* Wiley, 1991.

[28] S. Jegelka and J. Bilmes. Submodularity beyond submodular energies: Coupling edges in graph cuts. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 1897 –1904, june 2011.

[29] N. M. Josuttis. *The C++ standard library: a tutorial and reference.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[30] A. Krause. Sfo: A toolbox for submodular function optimization. *J. Mach. Learn. Res.*, 11:1141–1144, Mar. 2010.

[31] A. Krause. Tutorial on intelligent optimization with submodular functions, 2011.

[32] D. Kressner. A note on using compact wy representations for updating qr decompositions, 2009.

[33] H. Lin and J. A. Bilmes. Optimal selection of limited vocabulary speech corpora. In *INTERSPEECH*, pages 1489–1492. ISCA, 2011.

[34] K. Martin and B. Hoffman. *Mastering CMake: A Cross-Platform Build System.* Kitware Inc, 01 2003.

[35] B. Meyer. *Object-oriented software construction (2nd ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[36] S. Meyers. *Effective C++ : 55 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley Professional, third edition, May 2005.

[37] B. K. A. Om, D. Kressner, and E. S. Quintana-ortiz. Blocked algorithms for the reduction to hessenberg-triangular form revisited, 2008.

[38] J. Orlin. A faster strongly polynomial time algorithm for submodular function minimization. *Mathematical Programming*, 118:237–251, 2009. 10.1007/s10107-007-0189-2.

[39] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 3 edition, 2007.

[40] M. Püschel. How to write fast numerical code, 2008.

[41] A. Schrijver. A combinatorial algorithm minimizing submodular functions in strongly polynomial time. *J. Comb. Theory Ser. B*, 80(2):346–355, Nov. 2000.

[42] M. L. Scott. *Programming Language Pragmatics, Second Edition*. Morgan Kaufmann, Nov. 2006.

[43] J. Siek, L.-Q. Lee, and A. Lumsdaine. Boost graph library. http://www.boost.org/libs/graph/, June 2000.

[44] R. Steinmann. Applying the roofline model, 2012.

[45] P. Stobbe and A. Krause. Efficient minimization of decomposable submodular functions. In *Proc. Neural Information Processing Systems (NIPS)*, 2010.

[46] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[47] F. G. Van Zee, R. van de Geijn, and G. Quintana-Orti. Restructuring the QR algorithm for High-Performance application of givens rotations. Technical report, The University of Texas at Austin, Department of Computer Sciences, Oct. 2011.

[48] D. S. Watkins. *Fundamentals of Matrix Computations (3rd Edition)*. John Wiley & Sons, Inc, New Jersey, USA, 2010.

[49] P. Wolfe. Finding the nearest point in a polytope. *Mathematical Programming*, 1976.

List of Figures