OPTIMIZING THE GENERAL FLOYD-WARSHALL ALGORITHM

Florian Bütler, Manuel Hässig, Lasse Meinen, Roman Niggli

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

The Floyd-Warshall algorithm is a widely used way of solving the all-pairs-shortest-path problem. Due to its popularity, much research has been done on ways to optimize the algorithm. In addition, the Floyd-Warshall algorithm is proven effective at solving a wide range of other problems, by simply replacing its input type and basic operations.

In this work, we implement several known optimization techniques on the all-pairs-shortest-path Floyd-Warshall, and confirm their effectiveness. We also implement these techniques on two other instances of the Floyd-Warshall algorithm, namely transitive closure and all-pairs-widest-path. We evaluate the optimizations on these two instances and confirm that they work on Floyd-Warshall instances beyond all-pairs-shortest-path.

Finally, we present an autotuner that empirically finds good parameters for several optimization strategies, and generates functional, optimized code for all three implementations.

1. INTRODUCTION

The well-known Floyd-Warshall algorithm [1] for computing the all-pairs shortest path problem underlies a more general structure that solves several related problems. Some examples are the computation of the transitive closure of a graph [2, 3], the computation of the widest path [4], Kleene's algorithm to transform a nondeterministic automaton to a regular expression [5] or the inversion of a matrix using the Gauss-Jordan method. Lehmann [6] relates these algorithms using the notion of computing the *closure of a matrix* with respect to a closed semi-ring. The computation of the closure with the Floyd-Warshall method is a k-i-j nested loop where the computation in the innermost loop uses the operations of a particular closed semi-ring suitable to a given application.

Motivation. In the era without free speedup due to Moore's law where we have hit the memory, instruction level parallelism, and power walls, it is as important as ever to write optimized programs to fully take advantage of the power of today's computers. However, optimizing numerical code often leads to a large unreadable tangled mess of unrolled and tiled loops, and single-static-assignment style code. Moreover, writing programs with very large unrolling factors stops being just boring and becomes downright infeasible to do by hand.

The natural reaction to this is of course to automate the process of writing highly optimized code. Generating optimized programs using autotuners, which generate a program specifically optimized for a given machine configuration. However, algorithm specific autotuners need to be reimplemented for each new algorithm, which often is a significant undertaking.

However, with sufficient structural similarity between algorithms (e.g. identical loop structure, similar data representation, instruction latency and throughput) the same optimizations techniques should intuitively lead to good results for all similar algorithms. For example, consider different algorithms with nested loops where the inner loop performs different operations. An autotuner capable of unrolling loop in the loop structure of this class of algorithm can optimize the unrolling factors for all such algorithms.

In this work we aim to exploit the structural similarity between the specializations of the general Floyd-Warshall algorithm using different closed semi-rings to apply the same optimizations to a whole class of algorithms.

Contribution. In this work we optimize implementations of the shortest-path, transitive closure and widest path algorithm by applying unrolling, cache tiling, and vectorization. Further, we present an autotuner which finds locally optimal unrolling and tiling parameters for a given machine.

We aim to exploit the similarities between the algorithms given the common framework of a matrix closure by writing an autotuner that is able to apply these optimizations to all three algorithms, i.e. all three closed semi-rings. With this autotuner we demonstrate the potential to reuse a general enough autotuner on a class of algorithms.

Related Work. The closed semi-rings Lehmann introduces in [6] are a subcategory of Kleene-algebras. Kozen provides a survey [7] about Kleene-algebras, their applications and how closed semi-rings relate to them. For an approachable introduction to algorithms based on matrix closure consider [8, 9]. Further, Dolan [10] implements a host of algorithms based on the closure on closed semi-rings.

There have been several works to autotune single-core numerical computations, especially linear algebra subprograms. In his PhD thesis, Nelson [11] provides a good summary of past efforts. Related to the all-pairs shortest path problem, the Han, Franzetti, and Püschel [12] created an autotuner for the Floyd-Warshall algorithm and applied unrolling and cache tiling, notably removing the k-loop dependency based on techniques from [13, 14].

2. BACKGROUND ON WARSHALL-FLOYD-KLEENE'S ALGORITHM

In this section, we give short definition of closed semi-rings, define the structure of the generalized standard Floyd-Warshall (FW) algorithm, present the two alternative versions we use and perform a simple cost analysis.

Closed semi-rings. A closed semi-ring is an algebra $\langle R, +, \cdot, *, 0, 1\rangle$, where R is a set, $+ : R \times R \to R$ and $\cdot : R \times R \to R$ are binary operations called addition and multiplication, resp., $* : R \to R$ is a unary operation called closure, and $0 \in R$, $1 \in R$ are constants. The algebra must fulfil the following axioms:

- $\langle R, +\rangle$ is a commutative monoid with identity element 0:
  - $a + (b + c) = (a + b) + c$
  - $a + b = b + a$
  - $a + 0 = a$
- $\langle R, \cdot\rangle$ is a monoid with identity element 1:
  - $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
  - $1 \cdot a = a = a \cdot 1$
- Multiplication distributes over addition:
  - $a \cdot (b+c) = a \cdot b + a \cdot c = (b+c) \cdot a = b \cdot a + c \cdot a$
- Closure satisfies the following equation: $a^* = 1 + a \cdot a^* = 1 + a^* \cdot a$

Lehman [6] shows that the Floyd-Warshall algorithm computes the closure of a matrix $A \in R^{N\times N}$ for any simple closed semi-ring over R.

We introduce specific examples in the next section. For simplicity, we omit $*$, 0 and 1 from the algebra definitions, as all three can easily be deduced from $\langle R, +, \cdot\rangle$.

Generalized FW algorithm. The generalized standard Floyd-Warshall algorithm can be seen in Alg. 1. It consists of three nested loops, where the outermost loop updates each entry in the $N \times N$ matrix C N times, and *cannot* be reordered with the two inner loops.

For a particular semi-ring $\langle R, +, \cdot\rangle$, we create an instance of the algorithm by setting `ADD` to be the additive operation $+$, `MUL` to be the multiplicative operation $\cdot$, and `C` to be such that $C \in R^{N\times N}$. For example, the generalized FW algorithm instantiated with the semi-ring $\langle \mathbb{R}^\infty, \min, +\rangle$, where `C` is the *cost matrix* of the underlying graph, solves the all-pairs shortest path (APSP) problem.

A comparison of the two algorithms reveals that the structure of the FW algorithm is very similar to that of Matrix-Matrix Multiplication (MMM). Due to cache locality, the loop order $k - i - j$ would to be changed to $i - j - k$, in order to achieve peak performance as is the case for MMM. However, in the case of FW, the iterations of the outer loop are not independent, that is, such a reordering without any other transformations would result in an incorrect version of the algorithm.

Algorithm 1 Structure of the generalized Floyd-Warshall algorithm.

```
1: function FLOYDWARSHALL($C, N$)
2:     for $k \leftarrow 1$ to $N$ do
3:     for $i \leftarrow 1$ to $N$ do
4:     for $j \leftarrow 1$ to $N$ do
5:         $C[i][j] \leftarrow$ ADD($C[i][j]$, MUL($C[i][k]$, $C[k][j]$))
```

Iterative FW algorithm: FWI. Han, Franchetti and Püschel [12] introduced a tiled and unrolled version of the FW algorithm for the APSP problem. For simplicity, we repeat its definition here for the generalized version, which can be seen in Alg. 2. The generalized FWI algorithm introduces two variations on the standard FW algorithm. First, to facilitate tiling in later optimizations, we now iterate over matrices `A`, `B` and `C`. The matrix `A` contains the distances from `i` to `k`, `B` those from `k` to `j`, and `C` those from `i` to `j`. Second, the two innermost loops are unrolled by factors U_i and U_j, respectively.

We further introduce a special version of FWI, algorithm 3 FWIabc, which assumes that matrices `A`, `B` and `C` *do not* alias. Under this assumption, we can reorder the three loops. Thus, we can now additionally unroll the `k`-loop.

For simplicity, we assume the unrolling factors U_i, U_j, U_i', U_j', U_k' divide the matrix size N.

Algorithm 2 FWI parameterized by unrolling parameters U_i, U_j and semi-ring parameters `ADD`, `MUL`.

```
1: function FWI($A, B, C, N$)
2:     for $k \leftarrow 1$ to $N$ do
3:     for $i \leftarrow 1$ to $N$ do
4:     for $j \leftarrow 1$ to $N$ do
                                  ▷ Loops below are completely unrolled
5:         for $i' \leftarrow i$ to $i + U_i - 1$ do
6:         for $j' \leftarrow j$ to $j + U_j - 1$ do
7:             $C[i'][j'] \leftarrow$ ADD($C[i'][j']$, MUL($A[i'][k]$, $B[k][j']$))
```

Tiled FW algorithm: FWT. Building on the previously introduced algorithms, Han, Franchetti and Püschel [12] introduced a singly cache-tiled version of the FW algorithm.

Algorithm 3 FWIabc parametrized by unrolling parameters U'_i, U'_j, U'_k and semi-ring parameters `ADD`, `MUL`.

```
1: function FWIabc($A, B, C, N$)                ▷ $A, B, C$ do not alias
2:     for $i \leftarrow 1$ to $N$ do
3:     for $j \leftarrow 1$ to $N$ do
4:     for $k \leftarrow 1$ to $N$ do
                                  ▷ Loops below are completely unrolled
5:         for $i' \leftarrow i$ to $i + U'_i - 1$ do
6:         for $j' \leftarrow j$ to $j + U'_j - 1$ do
7:         for $k' \leftarrow j$ to $k + U'_k - 1$ do
8:             $C[i'][j'] \leftarrow$ OP1($C[i'][j']$, OP2($A[i'][k']$, $B[k'][j']$))
```

We again state its definition here for the generalized version. The algorithm 4 FWT iterates over tiles of size $L1 \times L1$, proceeding in four phases. In the first phase, the diagonal tile C_{kk} is updated. In the second and third phases, the tiles in the column and row of C_{kk} are updated. In the fourth phase, all remaining tiles are updated. The three tiles under consideration are guaranteed to be independent in the fourth phase and we can therefore use the `FWIabc` subroutine. Note that the fourth phase encompasses the bulk of the computation. Therefore, this separation of the algorithm into four phases allows us to run the majority of the computation without the aforementioned outer-loop dependency.

For simplicity, we assume the tile size $L1$ divides the matrix size N.

Algorithm 4 FWT parameterized by tile size $L1$, and the parameters for `FWI` and `FWIabc`

```
1: function FWT($A, B, C, N, L1$)
2:     // $A_{ij}$: $L1 \times L1$ submatrix $(i, j)$ of $A$
3:     $M \leftarrow N/L1$
4:     for $k \leftarrow 1$ to $M$ do
5:         FWI($A_{kk}, B_{kk}, C_{kk}, L1$)                ▷ Phase 1
6:         for $j \leftarrow 1$ to $M, j \neq k$ do
7:             FWI($A_{kk}, B_{kj}, C_{kj}, L1$)            ▷ Phase 2
8:         for $i \leftarrow 1$ to $M, i \neq k$ do
9:             FWI($A_{ik}, B_{kk}, C_{ik}, L1$)            ▷ Phase 3
10:        for $i \leftarrow 1$ to $M, i \neq k$ do
11:            for $j \leftarrow 1$ to $M, j \neq k$ do
12:                FWIabc($A_{ik}, B_{kj}, C_{ij}, L1$)     ▷ Phase 4
```

Cost Analysis. We define the cost of the FW algorithm to be the total number of operations on the semi-ring, that is, the number of `ADD` and `MUL` operations. Note that if the additive and multiplicative operations are not equally expensive in terms of the number of CPU cycles on a particular system, this cost measure will not accurately represent the performance of an implementation. In our case, however, this does not pose a problem, as the two operations are equal in cost for all three FW instances under consideration.

The general FW algorithm consists of three nested loops with two operations in the innermost loop and therefore we have $2N^3$ operations according to our cost function. Thus, it is trivial to see that the asymptotic complexity of the FW algorithm is $\mathcal{O}(N^3)$. As this report does not deal with instances of the FW algorithm for sparse matrices, the upper asymptotic bound simplifies to a tight one.

3. METHOD

In this section, start with an overview of the FW instances we examine, and the infrastructure we work with. Then, we briefly describe our process of writing baseline implementations, and then implementing various optimizations.

This work is focused on three particular semi-rings, each giving rise to its own instance of the Floyd-Warshall algorithm, targeted at solving its own problem.

- The semi-ring $\langle \mathbb{R}^\infty, \min, + \rangle$ corresponds to the all-pairs *shortest-path problem* (APSP). It computes the length of the shortest path between every pair of vertices in a graph.
- The semi-ring $\langle \mathbb{R}^\infty, \max, \min \rangle$ corresponds to the all-pairs *widest-path problem* (MM). It maximizes the smallest weight, a path between every pair of vertices has.
- The semi-ring $\langle \{0, 1\}, \vee, \wedge \rangle$ corresponds to the *transitive closure* (TC). It checks for every pair of vertices, if one is reachable from the other.

For each of these three semi-rings, we start by writing simple reference implementations in *Python* and *Go*. These are not used for benchmarking purposes, but to ensure our later implementations are correct and produce the same results.

Next, we write and benchmark a naive baseline implementation for each semi-ring. Due to the algorithm's simplicity, many of the more standard optimization techniques like strength reduction or function inlining are not possible. For this reason, we optimize FW by implementing the previously introduced `FWI` and `FWT` algorithms for the three listed FW instances. Additionally, we implement vectorized versions of the resulting 6 implementations.

Lastly, we implement and evaluate an autotuner, which finds optimal parameters and generates code for `FWI`, `FWT`, and the corresponding vectorized versions.

Setup. Figure 1 illustrates our project setup. We generate input graphs with *Python* and the *networkx* library. Using our reference implementations, we compute correct outputs for each testcase. To ensure our optimized implementations are correct, we run them against the same testcases and compare the results with the reference results. Finally, we have a separate set of larger inputs, against which we benchmark the validated, optimized implementations.

Since boolean values require much less memory space than floating-point numbers, benchmarking the TC implementations requires much larger inputs than APSP and MM, which both operate on double-precision floating-point numbers. Hence, storing input graphs for transitive closure the same way as for the other two problems would have been comically impractical - the larger graphs would have required several GB of space each. To circumvent this issue, transitive closure implementations do not read any input when benchmarking for large input sizes. Instead, they simply allocate sufficient space to store the matrix to operate on, but do not overwrite it. This essentially provides random input data, which is perfect for our benchmarking purposes since the algorithm runtime is not dependent on the input values.

Please note that the reason this trick is possible for transitive closure, but not for the other two problems, is that computing the transitive closure on boolean values has no additional requirements on the underlying graph. In comparison, APSP requires that the underlying graph does not have any *negative cycles*. Of course, this makes sense: If a negative cycle exists in a graph, any connection can be routed through that cycle infinitely often, producing a shortest distance of $-\infty$.

But beyond that, floating-point numbers also have a range of possible ways to represent values beyond numbers in $\mathbb{R}^{\infty}$. To make sure, the APSP and MM implementations are always benchmarked on well-defined and suitable graphs, they are always run against previously generated matrices.

To make all builds reproducible, implementations are compiled in docker. This ensures that compiler versions are consistent over all tested implementations.

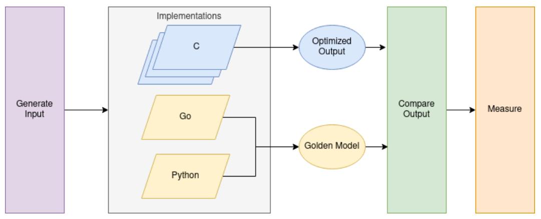

Fig. 1: The Building, Testing and Benchmarking System

Naive Implementation. For APSP and MM, the naive implementation is as simple as possible, looking almost exactly like 1. However, the naive implementation for TC already includes one optimization. The smallest datatype which C allows one to operate on is `char`. It has a size of 1 byte, or 8 bits. But each of these bits is already enough to store a boolean value. The naive TC therefore already performs *bit-packing*. That is a process wherein each byte accounts for 8 boolean values instead of just one. This not only vastly reduces the space required to store and operate on matrices, but it also allows us to perform 8 logical operations at once, since taking the conjunction between two bytes actually performs eight bitwise conjunctions.

This is very similar to the later optimization of vectorization, as it essentially operates of vectors of 8 booleans each.

Loop Unrolling. Perhaps the most simple optimization performed in this project is *loop unrolling*. Starting with the naive implementation, the two inner loops can be arbitrarily unrolled. But as mentioned earlier, the outermost loop cannot be unrolled, as it would violate dependencies.

In an additional, non-trivial optimization, the load of `C[i, k]` is moved out of the innermost loop. Furthermore, this value is not reloaded for the entire run of this loop. At first, this optimization appears to be faulty, as the innermost loop can change `C[i, k]` if `j` is equal to `k`. However, in that case the update for APSP has the form of `C[i, k] = MIN(C[i, k], C[i, k] + C[k, k])`. Since the algorithm requires the input graph not to have any negative circles, `C[k, k]` cannot be negative. Hence, `C[i, k] + C[k, k]` will never be smaller than just `C[i, k]`, meaning the update in question will never actually modify the value of `C[i, k]`.

For MM and TC, this optimization does not even require the input to fulfil any non-trivial problems. For MM, the update has the form `C[i, k] = MAX(C[i, k], MIN(C[i, k], C[k, k]))`, and for the TC it is `C[i, k] = C[i, k] | (C[i, k] & C[k, k])`. In both cases, it is easy to see that the value of `C[i, k]` is left unchanged.

For easier understanding, a concrete example of unrolled code can be found in the appendix A.1.

SIMD Vectorization. While certainly sounding more interesting, the introduction of SIMD (single instruction, multiple data) vector instructions is similar to just unrolling the innermost loop, but without all the additional load, compute and store statements. The unrolling factor depends on the data type the algorithm works with, as well as the hardware the program is to be run on. For this project, we work with the `AVX` and `AVX2` vector instruction set, meaning the vector instructions work with vectors of 256 bits. For the APSP and MM implementations, one such vector holds four doubles, meaning the innermost loops are essentially unrolled by a factor of four. Regarding the TC, one such vector can hold 32 chars. This means that when compared to the naive implementation, the vectorized version unrolls the innermost loop by a factor of 32.

Tiling. This optimization is explained more thoroughly in section 2 and Alg. 4. But in brief, the idea is to cut the matrix in tiles, and iterate over them. This not only allows for a faster loop ordering on some tiles, but if the tiles are of a suitable size, they also fit into the cache. As a result, the processor has to spend significantly less time waiting for data, allowing it to more efficiently utilize its resources.

In addition to tiling, these implementations also have loop unrolling. In fact, using them with the tile size set to the entire input matrix is identical to using the unrolled version.

The tiled implementations are all parameterized by the tile size, which is set as high as possible such that three tiles fit into the machine's L2 cache. To make the implementations more simple, we require the tile size divide the dimension of the input matrix, or the number of vertices in the input graph. Furthermore, the tile size also has to be a multiple of the *unrolling factor* of the functions operating on the tiles.

Vectorized Tiling. After the combination of tiling and unrolling, the obvious next step is to add vectorization too. As mentioned, vectorization is very similar to loop unrolling. The constraints on the tile size are therefore left unchanged for both the APSP and MM implementations. However, in the case of TC, vectorization means that the unrolling factor becomes 256. Coupled with the two constraints of dividing the input size and fitting into the cache, the vectorized tiled TC implementation is often forced to use significantly smaller tile sizes than would be optimal, leading to notable performance hits. For a more stable implementation, the input matrix could be extended with zeros to allow for an optimal tile size. However, such an optimization is not part of this work.

Autotuner. In spite of the aforementioned constraints, there still is a vast number of possible parameters with which to implement a particular optimization, or a combination thereof. Testing them all in order to find the best ones is therefore highly unpractical. In addition, parameters that are optimal for one input size may not be the best for another size. In order to find good parameters without tedious manual testing, we implement an autotuner, capable of testing parameters and searching good ones on its own. While this autotuner is still unable perform an exhaustive search over all possible parameters, it can find locally optimal parameters and produce according code.

Figure 2 shows the process by which the autotuner formulates its guesses on suitable parameters, and then tries to refine them. It starts off by trying all possible parameters for an input size small enough to do so in reasonable time ($N = 64$ in our case). On the larger input size, the autotuner employs a hill-climbing algorithm to search for locally optimal parameters using the optimal parameter the algorithm found for a small input as an initial guess. This algorithm again follows the illustrated process, where each round begins with the computation of parameter sets to try next. These parameter sets are generated by changing only one parameter of the current set, while leaving the rest unchanged. The new value for the changed parameter is always a neighbor of the current value, meaning either the next larger or next smaller value while respecting existing constraints. In each round, the autotuner evaluates the performance on each of the generated parameter sets. If one of them surpasses the current best, the autotuner uses the according parameters as basis for the next round. And if it cannot find better parameters, it terminates, as a local optimum has been found.

Upon termination, the autotuner returns the locally optimal parameters it found, in addition to accordingly generated code. It works for each of the three semi-rings examined in this work. It can either perform unrolling, where it optimizes the unrolling factors, or it can perform both unrolling and tiling, where it optimizes the tile size and the unrolling factors. Furthermore, both versions support vectorization.

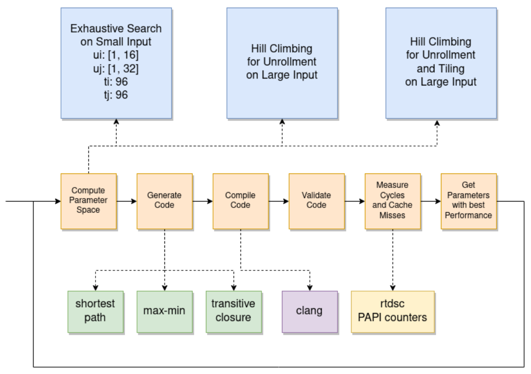
Exhaustive Search on Small Input
ui: [1, 16]
uj: [1, 32]
ti: 96
tj: 96
Hill Climbing for Unrollment on Large Input
Hill Climbing for Unrollment and Tiling on Large Input
Compute Parameter Space
Generate Code
Compile Code
Validate Code
Measure Cycles and Cache Misses
Get Parameters with best Performance
shortest path
max-min
transitive closure
clang
rtdsc
PAPI counters

Fig. 2: Autotuning Method

4. EXPERIMENTAL RESULTS

In this section we describe our experimental setup, and describe and interpret our results for the three different algorithm instances. The results will compare the naive implementation to the autotuned versions of FWI and FWT both as scalar and vectorized implementations.

Experimental setup. Due to time constraints, the experiments were run on three different platforms, one per algorithm instance. The respective platform specifications can be seen in Tbl. 1. For brevity, the CPU brands are omitted, as all three are Intel CPUs. β is a shorthand notation for memory bandwidth in bytes/cycle.

The source code implementations were compiled with `clang` version 13.0.1. Vector code was compiled with `-O3 -ffast-math -march=native`, scalar with `-O3 -no-loop-unroll -fno-slp-vectorize`.

Please note that we intentionally added the compiler flags `-no-loop-unroll -fno-slp-vectorize` and omitted the compiler flags `-ffast-math -march=native` for scalar code to ensure the compiler did not perform unrolling and vectorization optimizations by itself, thereby confounding our performance measurements. We performed a small search for different compilers and compiler flags to ensure we used the best-possible compiler and flags.

The compiled implementations for APSP and MM were run and measured on input sizes $N \in [432, 4608]$, where the input matrix C consists of randomly generated double-precision floating-point values and has size $N \times N$. For TC, we have $N \in [256, 20992]$ and the input matrix C of size $N \times N$ consists of randomly generated bits. In any case, we collected enough measurements to ensure timing overhead was insignificant and little to no variation could be observed.

In order to obtain measurements for performance and amount of data transferred, we measured the number of cycles and the number of misses in the L3 cache using the PAPI library [15]. We warm up the cache to ensure proper measurements.

	APSP	MM	TC
CPU	Xeon E3	Core i7	Core i7
Base Freq.	2.8GHz	1.8GHz	1.8GHz
μarch	Skylake	Coffee Lake	Coffee Lake
SIMD	AVX-2	AVX-2	AVX-2
L3 size	8 MiB	8 MiB	12 MiB
β [B/cycle]	6.79	7.53	7.03

Table 1: Specifications of respective platforms

All-Pairs Shortest Path Results.
Figure 3 shows the observed performance for the APSP problem. For scalar code, the tiled FWT code achieves 3.3x speedup relative to the naive implementation and 110% of the theoretical peak scalar performance as there seems to be a bug in the autotuner which leads it to generate vectorized code for the scalar implementations. For vector code, the tiled FWT significantly outperforms the non-tiled counterpart and achieves 9.5x speedup relative to the naive implementation and 78% of the theoretical peak SIMD performance.

Interestingly, the tiled, non-vectorized code outperforms the non-tiled, vectorized code for larger input sizes. In fact, we see that the performance of the non-tiled, vectorized implementation converges to that of the non-tiled, non-vectorized implementation. We can explain this result using the roofline model [16]. Figure 4 shows the roofline plot for the APSP problem. Smaller input sizes are show with smaller dots and larger input sizes with larger dots. The plot shows that non-tiled implementations quickly become memory-bound while tiled implementations remain compute-bound.

Around $N = 2880$ we see a pronounced drop in performance for FWT-SIMD and less pronounced for the scalar FWT. There are a few possible explanations for this drop. One possibility is that the hill-climbing during the autotuning settled on a suboptimal local-maximum, which is possible due to the severe non-convexity of the parameter space. Another possibility is that the autotuner is limited in its choice of legal tile sizes due to the divisibility of the input size. The fact that 2880 is highly divisible (in fact even more divisible than the adjacent input sizes) speaks against the latter option. But the observation that the vectorized implementation is much more affected speaks for the latter possibility, as vectorization reduces the set of legal tile sizes.

All in all, we can see that the autotuner is able to successfully apply effective optimizations to the Floyd-Warshall algorithm.

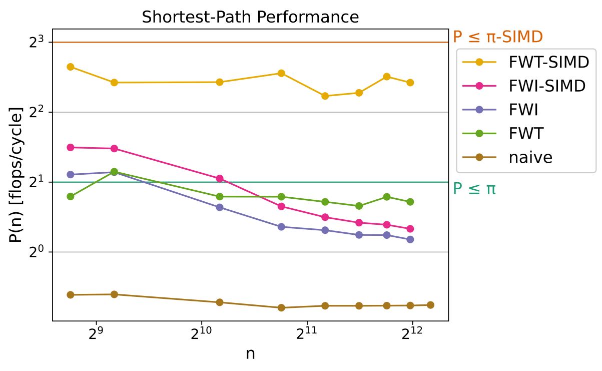

Fig. 3: Performance comparison of FW implementations for APSP.

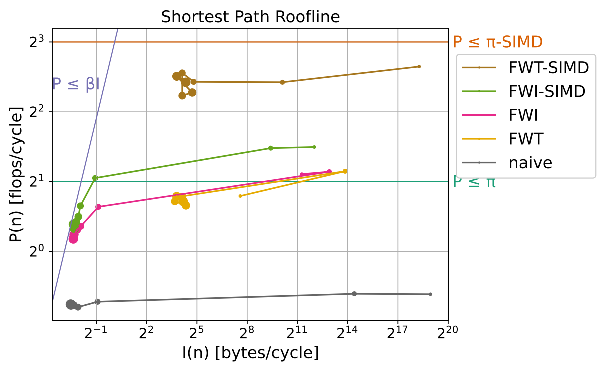

Fig. 4: Roofline plot of FW implementations for APSP.

Max-Min Results. As the structure of the computation, the instruction latency and throughput, as well as the data structure and representation is identical to that of the APSP instance, we expect our optimizations to reach similar performance values for the MM instance.

Figure 5 shows the performance of the MM implementations. For scalar code, the tiled FWT code achieves 2.2x speedup relative to the naive implementation and 103% of peak scalar performance because of the bug in the autotuner. For vector code, the tiled FWT significantly outperforms the non-tiled counterpart and achieves 6.1x speedup relative to the naive implementation and 72% of the theoretical peak SIMD performance.

Similarly to the APSP instance, the performance for both tiled implementations shows a dropoff at $N = 2304$. This is interesting as it would be odd for the autotuner to manifest the same quirk to find a suboptimal local maximum for the same region of input sizes. It is not impossible, however, as the performance for $N = 2880$ shows improvement which it did not do for the APSP instance.

Looking at the roofline plot in figure 6 we find the same characteristic for the MM instance that only tiled implementations manage to stay computationally bounded. This is an example of one autotuner being able to effectively apply the same optimizations to two different algorithms.

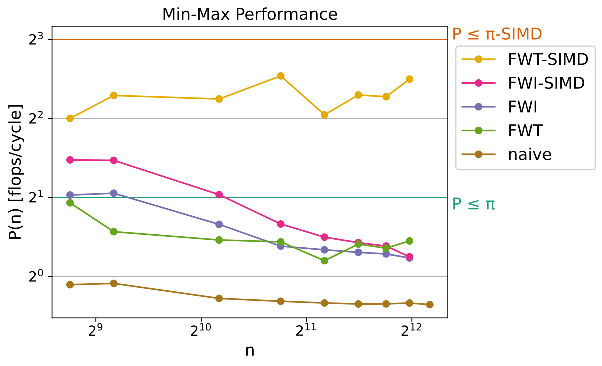

Fig. 5: Performance comparison of FW implementations for MM.

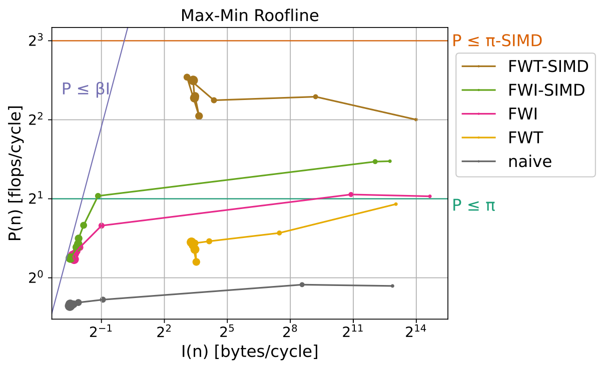

Fig. 6: Roofline plot comparing implementations for Max-Min on an Intel Core i7 (Coffee Lake microarchitecture).

Transitive Closure Results. The TC instance, while following the same algorithmic and data structure has a different data representation and instruction latency and throughput. As a result, it is not a given that the optimizations translate to this instance.

Figure 7 shows the performance plot for the TC instance. The scalar FWT implementation achieves 65% of the theoretical scalar peak performance. However, due to finding and fixing a bug too close to the deadline, we were not able to run the measurements for all input sizes as running the autotuner takes considerable amount of time, especially for the large input sizes. The vectorized FWT implementation achieves a 33.3x speedup compared with the naive implementation and 65% of the theoretical SIMD peak performance.

Note that the vectorized tiled implementation exhibits large, intermittent dropoffs in performance. In this case we know that this is caused by a constriction of the parameter space for certain input sizes leading to the optimal tiling factors not being available for selection by the autotuner. The lack of legal optimal tiling factors is due to the bit-packing, which packs 256 bits into a single vector register. Therefore, the number of available factors and therefore tiling factors diminishes considerably.

Looking at the roofline plot in figure 8 we see that, while both vectorized implementations initially display similar performance characteristics, the fact that, like for the other instances, only the tiled implementations remain compute-bound, the tiled implementation performs better for large input sizes.

This shows that the autotuner was still able to successfully apply the same optimizations and achieve a similar result even for different data representations and different instruction latencies and throughputs. However, the TC instance also shows limits to the generality of our autotuner. While the vectorized FWT remained compute bound, its performance was severely decreased as a consequence of the different data representation with the bit-packed vectors. However, the different instruction latencies and throughputs do not impact the performance, as an autotuner finds the optimal values empirically.

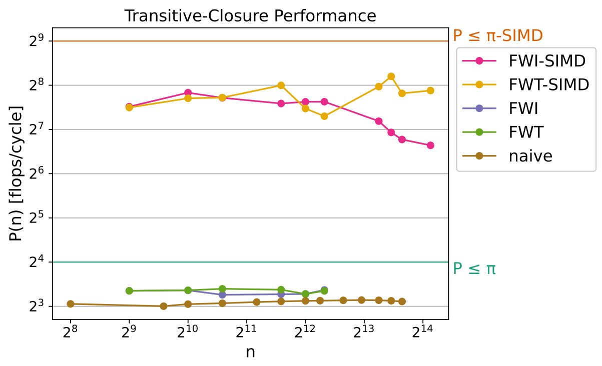

Fig. 7: Performance comparison of FW implementations for TC.

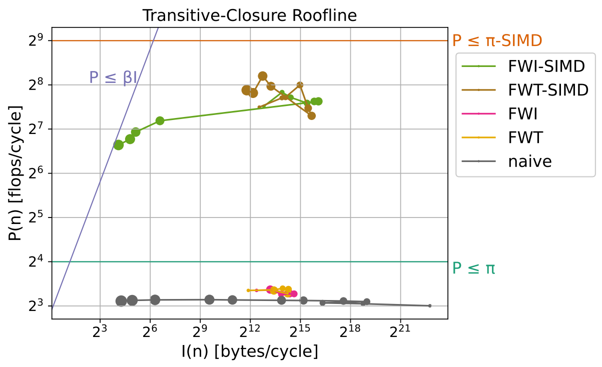

Fig. 8: Roofline plot comparing implementations for Transitive Closure on an Intel Core i7 (Coffee Lake microarchitecture).

5. CONCLUSIONS

To summarize, we examined three instances of the generalized Floyd-Warshall algorithm. Respectively, they solve the all-pairs shortest-path, widest-path and transitive closure problems. We implemented various optimizations for each, namely `FWI`, `FWT`, and corresponding vectorized versions. Finally, we implemented an autotuner that finds locally optimal unrolling and tiling factors and generates corresponding code.

Our benchmarks confirm that the optimization techniques for the all-pairs shortest-path problem by Han, Franchetti and Püschel [12] work, and we show that they generalize to the other two examined closed semi-rings of the widest-path and transitive closure instances. Further, we showed that a single autotuner implementation is able to successfully apply the same optimizations to all three problems.

However, we also found limits to the generality of our autotuner as soon as the data representation changes. This leads us to conjecture that our autotuner should be easily generalizable to other closed semi-rings over $\mathbb{R}^{\infty}$.

Future Work. An obvious next step would be to implement other closed semi-rings in our autotuner. A good choice would be the unpivoted Gauss-Jordan transform, as it is a semi-ring over $\mathbb{R} \cup \{\text{undefined}\}$. For further generalization we would likely need an autotuner which is aware of the program structure and is able to apply transforms based on the structure as needed for different data representations. Ideally, one would implement an autotuner fully generic over a closed semi-ring.

There are also more optimizations that could be applied to the Floyd-Warshall algorithm. One example is the doubly tiled version described by Han et al. [12]. Another possible optimization would be to explore different algorithms for matrix closure. One possible candidate might be R-Kleene [17].

6. CONTRIBUTIONS OF TEAM MEMBERS

In this section we briefly list what each team member contributed to the project in terms of code written for optimization and analysis. Please note that while infrastructure- and testing-related code is not included here, props go out to Manuel for setting up an awesome build system utilizing Docker containers to ensure identical dependencies and to Florian for setting up a highly-convenient, all-powerful development pipeline.

Florian. I built the foundation of the autotuner including generic code templates for all three algorithms with cache tiling, loop unrolling and vectorization. The autotuner is initially using an exhaustive search and then applies hill climbing to find the optimal parameters.

Roman. I added bit-packing to the transitive closure infrastructure by adjusting the code to read and write input, and the naive implementation. I wrote the initial vectorized versions for all three problems. I helped in applying other optimizations to transitive closure, and fixing some of the problems that arose in the process.

Manuel. I set up the measurements using PAPI counters needed for the plots and the autotuner. I polished Florian's initial autotuner implementation with an improved hill climbing algorithm. Further, I applied and fine-tuned optimizations for the shortest path implementation.

Lasse. I implemented vectorized and non-vectorized versions of the FW, FWI, FWIabc and FWT algorithms for the widest-path, that is, Max-Min, algorithm. I supported Manuel and Roman in applying FWI and FWT optimizations to their programs and built on Manuel's and Florian's work to integrate FWT optimizations and vectorization in the final version of the autotuner.

7. REFERENCES

[1] Robert W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, pp. 345, jun 1962.

[2] Stephen Warshall, "A theorem on boolean matrices," *J. ACM*, vol. 9, no. 1, pp. 11–12, jan 1962.

[3] Bernard Roy, "Transitivité et connexité," *Comptes Rendus Hebdomadaires Des Seances De L Academie Des Sciences*, vol. 249, no. 2, pp. 216–218, 1959.

[4] Maurice Pollack, "Letter to the editor—the maximum capacity through a network," *Operations Research*, vol. 8, no. 5, pp. 733–736, 1960.

[5] S. C. Kleene, *Representation of Events in Nerve Nets and Finite Automata*, pp. 3–42, Princeton University Press, 1956.

[6] Daniel J. Lehmann, "Algebraic structures for transitive closure," *Theoretical Computer Science*, vol. 4, no. 1, pp. 59–76, 1977.

[7] Dexter Kozen, "On kleene algebras and closed semirings," in *Mathematical Foundations of Computer Science 1990*, Branislav Rovan, Ed., Berlin, Heidelberg, 1990, pp. 26–47, Springer Berlin Heidelberg.

[8] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley series in computer science and information processing. Addison-Wesley Publishing Company, 1974.

[9] Kurt Mehlhorn, *Path Problems in Graphs and Matrix Multiplication*, pp. 133–170, Springer Berlin Heidelberg, Berlin, Heidelberg, 1984.

[10] Stephen Dolan, "Fun with semirings: A functional pearl on the abuse of linear algebra," *SIGPLAN Not.*, vol. 48, no. 9, pp. 101–110, sep 2013.

[11] Thomas Nelson, "Dsls and search for linear algebra performance optimization," 2015.

[12] Sung-Chul Han, Franz Franchetti, and Markus Püschel, "Program generation for the all-pairs shortest path problem," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, New York, NY, USA, 2006, PACT '06, p. 222–232, Association for Computing Machinery.

[13] Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya, "A blocked all-pairs shortest-paths algorithm," in *Algorithm Theory - SWAT 2000*, Berlin, Heidelberg, 2000, pp. 419–432, Springer Berlin Heidelberg.

[14] J.-S. Park, M. Penner, and V.K. Prasanna, "Optimizing graph algorithms for improved cache performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 9, pp. 769–782, 2004.

[15] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, Eds., Berlin, Heidelberg, 2010, pp. 157–173, Springer Berlin Heidelberg.

[16] Samuel Williams, Andrew Waterman, and David Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, apr 2009.

[17] Paolo D'Alberto and Alexandru Nicolau, "R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks," *Algorithmica*, vol. 47, no. 2, pp. 203–213, Feb 2007.

A. APPENDIX

A.1. Code Samples

```
int floydWarshall(double *C, int N) {
  for (int k = 0; k < N; k++) {
    for (int i = 0; i < N; i ++) {
      double cik = C[i * N + k];
      for (int j = 0; j < N - 1; j += 2) {
        // load
        double cij0 = C[i * N + j + 0];
        double cij1 = C[i * N + j + 1];

        double ckj0 = C[k * N + j + 0];
        double ckj1 = C[k * N + j + 1];

        // compute 1
        double sum0 = cik + ckj0;
        double sum1 = cik + ckj1;

        // compute 2
        double res0 = MIN(cij0, sum0);
        double res1 = MIN(cij1, sum1);

        // store
        C[i * N + j + 0] = res0;
        C[i * N + j + 1] = res1;
      }
    }
  }
  return 0;
}
```

Listing 1: An unrolled implementation of the APSP FW instance.

A.2. Autotuner Parameter Choices

The following are the autotuner's choices of unrolling factors and tile sizes, all stemming from the MM instance. In the tables, N denotes the input size, L_1 the tile size, and $U_{i,j,k}$ the unrolling factors of the various loops.

In the case of `FWT`, the unrolling factors $U_{i,j}$ concern the function `FWI`, and the unrolling factors $U'_{i,j,k}$ concern the function `FWIabc`.

N	U_i	U_j
432	3	1
576	3	1
1152	3	1
1728	3	1
2304	3	1
2880	3	1
3456	3	1
4032	3	1

Table 2: Parameters for scalar `FWI`

N	L_1	U_i	U_j	U'_i	U'_j	U'_k
432	144	3	1	1	9	16
576	576	3	1	1	8	18
1152	96	3	1	1	8	16
1728	144	3	1	1	8	18
2304	96	1	1	1	8	16
2880	192	2	1	1	6	4
3456	128	2	1	1	8	16
4032	224	2	1	1	8	16

Table 3: Parameters for scalar `FWT`

N	U_i	U_j
432	4	8
576	3	4
1152	8	4
1728	9	4
2304	8	4
2880	8	4
3456	4	12

Table 4: Parameters for vectorized `FWI`

N	L_1	U_i	U_j	U'_i	U'_j	U'_k
432	144	9	4	4	12	16
576	96	8	8	4	16	16
1152	96	8	4	4	8	16
1728	144	8	4	4	8	16
2304	96	4	4	3	16	16
2880	192	8	8	4	8	16
3456	128	8	4	4	16	16
4032	192	8	4	4	8	16

Table 5: Parameters for vectorized `FWT`