OPTIMIZING GENERALIZED FLOYD-WARSHALL

Andrew Dobis, Marie Jaillot, Michele Meziu, Shashank Anand

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Modern-day users expect an ever-increasing performance from the applications they use. To keep up with this demand, many algorithms must be implemented in a way which maximizes the performance on every potential system. Recent work has shown the potential of domain-specific program generators to automate the production of code which maximizes performance on an architecture-specific level. Using this approach, we propose an optimized implementation of the generalized Floyd-Warshall algorithm using program generation, vectorization, tiling, as well as auto-tuning to find the optimal parameters for a given system. We evaluate our implementation on a `3.6GHz` Intel `Coffee Lake` processor. The proposed optimized implementation yields a speedup of up to $3.1\times$ over the baseline, and roofline evaluations show that our performance comes within $2\times$ of the performance ceiling of this compute-bound problem.

1. INTRODUCTION

The current never-ending increase in the demand for high performance applications has lead to the exploration of many different methods to achieve these results. One such method is domain-specific code generation, which automates the process of creating a tailored implementation that maximizes the performance of a specific application on a given system.

Motivation. Many important applications rely on shortest-path computations in order to achieve their goal, such as navigation systems, financial arbitrage, or social network analysis [1, 2] to name a few. Performance is critical for these applications in order to maintain real-time constraints. For example, navigation systems require that the shortest-path to the requested destination be computed as fast as possible, as to avoid falling off the new path before the computation has terminated, thus yielding an invalid result.

The Floyd-Warshall algorithm is a popular solution to this problem. This can be used to compute the shortest path between all pairs of vertices in a given graph. While often used as a solution, a naive implementation often yields poor performance. However, achieving a high performance is non-trivial, since the inter-dependencies between different operands as well as the access patterns used in the the main computation make certain optimizations difficult to implement effectively. For these reasons, achieving a fast implementation of the generalized Floyd-Warshall algorithm will require many non-trivial system-specific optimizations.

Contribution. This work proposes a fast implementation of the generalized Floyd-Warshall algorithm. This is achieved using various optimizations such as tiling, and vectorization. Program generation and auto-tuning are used to create an adaptive solution, allowing for Instruction Level Parallelism (ILP) to be maximized. We then evaluate a generated implementation using auto-tuned tiling parameters on multiple different graphs of various sizes.

Related work. Existing works on this topic have often focused on parallel implementations rather than sequential ones. Lund and Smith [3] leverage the heterogeneity of most systems by implementing the algorithm as a CUDA kernel. While their solution relies on the use and availability of a GPU in order to run their implementation, we focus on single-core optimizations, using widely available superscalar hardware.

Another solution is to exploit the multi-core nature of modern processors. Zhang et al [4] propose such a solution, by using Intel's Thread Building Blocks (TBB) in order to create a parallel implementation of the algorithm. In contrast to this, our work focuses on maximizing single-core performance, rather than utilizing hardware which might not be available on every system.

Pluta [5] takes a similar approach to ours, proposing a minutely tuned assembly implementation of the algorithm which aims to manually achieve maximal ILP for a specific system in order to obtain an optimal performance. While this yields a good performance on one machine, it remains single-system-specific, while our solution focuses on being adaptive in order to maximize the single-core performance on any given system.

Finally, our implementation is based on the solutions proposed in Han, et al [6]. In this work the authors propose a program generation method which maximizes the single-core performance on a given system. Similar to this work, we also propose an adaptive solution which uses program generation in order to maximize the amount of ILP obtained on every potential system.

2. BACKGROUND

Before presenting our solution, we must first describe the problem we have set out to solve. We will start by presenting the internal graph representation which will be used, followed by the Floyd-Warshall algorithm itself, and finally a generalized version of said algorithm.

Graph Encoding. Before we can present the algorithm itself, it is important to define how we encode our graphs.

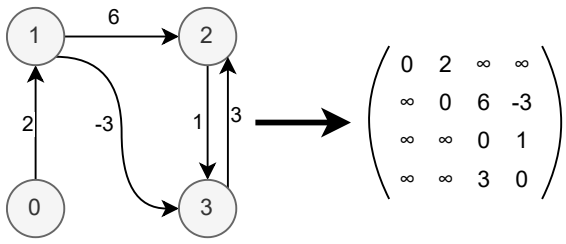

Fig. 1. Matrix encoding of a weighted graph for the all-pairs shortest path problem. The matrix represents the weights of each edge in the graph, where 0 is used to relate a vertex to itself and ∞ is used when no such edge exists.

Figure 1 shows an example representation of a graph using a weight matrix. Each element in the matrix represents an edge in the graph such that element $\{i, j\}$ of the matrix represents the weight w_{ij} of the edge between vertex i and vertex j. If $i = j$ then $w_{ij} = 0$ and if there is no edge from i to j, then $w_{ij} = \infty$ [6, 7]. This type of representation will be used for the input graph of our program. We will now define the Floyd-Warshall algorithm itself.

Floyd-Warshall Algorithm. First formalized in 1962 by Robert W. Floyd [8], the Floyd-Warshall algorithm can be used to efficiently compute the shortest path between all pairs of vertices within a given graph. The algorithm has the following signature:

- **input** $G = (V, E)$ with $V = \{1, 2, ..., n\}$ as weight matrix $C \in \mathbb{R}^{n \times n}$.
- **output** $D = (d_{ij}) : d_{ij} = \delta(i, j)$, where $\delta(i, j)$ is the weight of the shortest path from i to j.

The core idea of the algorithm is to compare the weights of each possible path between two vertices in a graph. However, doing so naively would lead to a complexity of $\Theta(n^4)$, since we would have to compare each pair of intermediate vertices with each other [7].

Floyd-Warshall manages to do this with a time complexity of $\Theta(n^3)$ [7] by exploiting the relationship between intermediate paths and the parent path they are part of. Given a shortest path computation from i to j where all intermediate vertices are in the set $\{1, 2, ..., k\}$ for some $k < n$, we have that, if k is not an intermediate vertex of the path from i to j, then we know that all of the intermediate vertices of that path are strictly inferior to k, meaning that a shortest path with all intermediate vertices in $\{1, 2, ..., k-1\}$ is also the one for intermediate vertices in $\{1, 2, ..., k\}$. In the case where k is an intermediate vertex of the path from i to j, then we can use the same relation as in the other case, but on the two sub-paths $i \longrightarrow k$ and $k \longrightarrow j$. This relation allows us to express the algorithm as a dynamic programming problem using the following recursive definition [7]:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

Using this recurrence relation, one can compute the shortest path between all pairs in a graph by only comparing intermediate paths which have the potential of being the shortest to one another, rather than comparing every possible paring of vertices.

Generalized Floyd-Warshall. Additionally to being useful for solving the all-pairs shortest path problem, the Floyd-Warshall algorithm can be generalized to solve other problems as well. In a general form, its recursive definition is as follows:

$$r_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \text{OP1}(r_{ij}^{(k-1)}, \text{OP2}(r_{ik}^{(k-1)}, r_{kj}^{(k-1)})) & \text{if } k \geq 1 \end{cases}$$

For a given semi-ring of the form $\{\texttt{OP1}, \texttt{OP2}\}$, this can be used to compute different results about the input graph. In fact, the algorithm proposed by Floyd [8] is based off of a work published two years prior which proposes an iterative method for performing the same computation using the Boolean $\{\texttt{OR}, \texttt{AND}\}$ operators [9]. Such a computation results in the transitive closure of the given graph[1] [10]. Alternatively, using the $\{\texttt{MAX}, \texttt{MIN}\}$ semi-ring results in an all-pairs widest path computation. All three of these semi-rings will be evaluated throughout this work.

Cost Analysis. In order to analyze the cost of this computation, we will be looking at the number of floating point or bit-wise operations, given that for each evaluated semi-ring, the two operations used have the same latency and throughput on the system we used for our benchmarks. An iterative definition of the computation is used to evaluate its cost:

for $\{k, i, j\}$ **in** $\{[0..n], [0..n], [0..n]\}$:
$C[i][j] = \textbf{OP1}(C[i][j], \textbf{OP2}(C[i][k], C[k][j]))$

Here, we can see that each iteration comprises of 2 operations, which are done in a triple for loop, yielding a final op-count of $2n^3$ operations in total.

[1]The transitive closure of a graph gives us information about the existence of a path connecting each vertex to another.

3. A FAST FLOYD-WARSHALL IMPLEMENTATION

Now that the algorithm has been defined, we can present how we implement an high performance version of Floyd-Warshall. In the following paragraphs, we present our baseline implementation of the algorithm followed by the different optimizations that were done in order to achieve our final high performance implementation. These include tiling, allowing the removal of dependencies between the different operands, blocking, in order to improve the spatial locality of the data being operated on, and finally AVX-2 vectorization, allowing for the exploitation of super-scalar hardware. Additionally, we will present our methods for auto-tuning and program generation, allowing our solution adapt to each system. Overall, our work contains fast implementations of the Floyd-Warshall algorithm over three different semi-rings, i.e. $\{$`MIN`, +$\}$, $\{$`MAX`, `MIN`$\}$, and $\{$`OR`, `AND`$\}$.

Baseline Implementation. Before any optimizations can be made, we need to look at our baseline. As briefly presented in the previous section, we will be working with an iterative definition of our algorithm. This will enable all of our future optimizations. The implementation itself will be very similar across all three semi-rings. In order to translate our recursive definition to an iterative one, it is important to maintain a coherent ordering of our indices, meaning that a k, i, j ordering must be used, since k is our recursion index and i, j are the indices of our resulting matrix. Existing work shows that this ordering must only be maintained if the operand matrices are non-distinct [6], which will come into play when we talk about tiling. Thus our baseline implementation will be as follows:

```
for (size_t k = 0; k < n; k++) {
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            C[i*n + j] = op1(C[i*n + j],
                op2(C[i*n + k], C[k*n + j]));
        }
    }
}
```

Note that the type used to represent our weights will differ depending on the semi-ring. For $\{$`MIN`, +$\}$, and $\{$`MAX`, `MIN`$\}$ a double-precision floating point array is used, while for $\{$`OR`, `AND`$\}$ 64-bit unsigned integers are used. The matrix itself is stored in row-major order.

Tiling and Blocking. Memory bandwidth analysis on the baseline shows a significant amount of cache misses, far more than the number of compulsory ones. Moreover, the inter-operand dependencies in the computation limit the use of Matrix-Matrix-Multiplication (MMM) style optimizations [11], which would otherwise be possible given the structure of the computation. We start by addressing the first problem, which is the low data locality exploitation. One solution to this is to use a form of blocking, which splits up the computation in order to work on small blocks data that fit into the cache blocks [12]. This greatly improves the spatial locality of the computation.

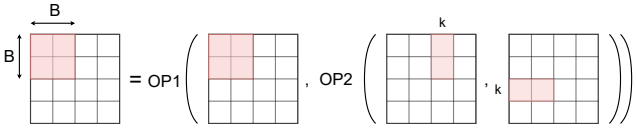

Fig. 2. Illustration of the blocking pattern for our generalized computation. In this example $B = 2$, $0 \leq i, j \leq 1$, $k = 2$.

Figure 2 illustrates how blocking would improve the locality of the data used, since the computation would then take place on a per-block level, where ideally all data would fit in a cache block, thus minimizing the cache miss rate. B represents the block size, i.e. $B \times B$ is the number of elements within a single block. One constraint is that B must divide n in order to avoid stray elements which do not fit in a block. The implementation of blocking is done by adding two inner-loops to the computation:

- Loop $ip \in [i, (i + B)]$, where ip will replace i.
- Loop $jp \in [j, (j + B)]$, where jp will replace j.

Due to the aforementioned inter-operand dependencies, blocking can not take place along the k loop.

In order to solve these dependencies, the structure of the computation can be modified such that sections of the operand matrices, i.e. tiles, are handled in different phases. This allows us to isolate the case where aliasing is no longer possible between the operands, thus enabling loop reordering in this newly create phase. This method is called tiling, and our implementation of it is strongly inspired by the one described in an existing work [6].

Figure 3 illustrates the access pattern used in the different phases of our tiling process. The idea is to cover all possible access patterns with these four phases. Phase 1 refers to the case where all three operands are identical. Phases 2 and 3 have two distinct operands, and phase 4 uses three distinct matrices as operands. In this phase, we are able to reorder the loops in order to have a i, j, k ordering, improving the locality of the access pattern, and allowing for blocking to occur along the k loop.

Algorithm 1 shows how the different phases are used together to achieve the final result. Note that as of this point, we consider our three operands to be separate matrices with potential aliasing, rather than three times the same matrix. Here, every phase does a blocked Floyd-Warshall computation on sub-matrices defined as $C_{ij} = C[i..(i+L1)][j..(j+L1)]$. In our implementation, these sub-matrices are used by simply taking $L1$ as a parameter and accessing the matrix using our defined pattern. Using this optimization, we

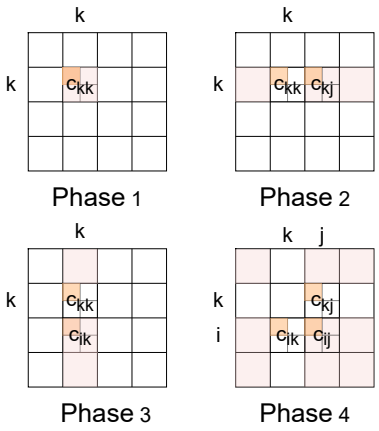

Fig. 3. The four phases of tiling. The pink shaded tiles are the ones that are modified in a given phase. For certain tiles, blocks are shown. The orange squares show corresponding blocks in the computation.

can greatly improve the locality exploitation in our computation. Additionally, this enables vectorization, which has the potential to greatly improve our instruction throughput.

AVX-2 Vectorization. In order to vectorize our implementation, we first select the index along which a loop-unrolling enables an efficient use of vector instructions. By analyzing the access pattern in our tiling phases, we find that the best loop to unroll is the inner jp loop, which we unroll by 4, given that AVX-2 has 256 bit vectors. This allows for the conversion of both $C[i][j]$, and $B[k][j]$ into vectors for our operation. The implementation itself differs between each semi-ring only by the core operational intrinsics used. We first create a vector with our non-vectorizable element $A[ip][kp]$ in the inner ip loop. In the unrolled jp loop, the vectors $C[ip][jp..(jp+4)]$ are first loaded, then operated on using the following intrinsics, depending on the semi-ring used:

- {`MIN, +`}: `_mm256_min_pd`, and `_mm256_add_pd`.
- {`MAX, MIN`}: `_mm256_max_pd`, and `_mm256_min_pd`.
- {`OR, AND`}: `_mm256_or_si256`, and `_mm256_and_si256`.

Finally we store the result back into $C[ip][jp..(jp+4)]$.

Auto-tuning. All of the aforementioned optimizations depend on user-defined parameters such as the tile size $L1$ or the block size B. The choice of these parameters impacts the performance of our program, and thus it is important to select the best best parameters for every tested matrix size in order to achieve a maximum performance. In order to guarantee the use of optimal parameters for our benchmarks, we implement an auto-tuning system, which finds the best parameters over a set of matrix sizes.

Algorithm 1 Tiled version of Floyd-Warshall

Require: $A, B, C \in \mathbb{R}^{n \times n}$
1: $m \leftarrow \frac{n}{L1}$
2: **for** $k \leftarrow [0..m]$ **do**
3: fw_phase_1(A_{kk}, B_{kk}, C_{kk}, L1)
4: **for** $j \leftarrow [0..m]; j \neq k$ **do**
5: fw_phase_2(A_{kk}, B_{kj}, C_{kj}, L1)
6: **end for**
7: **for** $i \leftarrow [0..m]; i \neq k$ **do**
8: fw_phase_3(A_{ik}, B_{kk}, C_{ik}, L1)
9: **end for**
10: **for** $i \leftarrow [0..m]; i \neq k$ **do**
11: **for** $j \leftarrow [0..m]; j \neq k$ **do**
12: fw_phase_4(A_{ik}, B_{kj}, C_{ij}, L1)
13: **end for**
14: **end for**
15: **end for**

Throughout our auto-tuning process, a few constraints must be taken into account. First, the matrix sizes used must not be incremented using a two-power stride in order to avoid an under-utilization of the different sets in our cache [12, 13]. Additionally, the tiling and blocking factors must divide each-other, meaning that $n \mod L1 = 0$ and $L1 \mod B = 0$, in order to avoid stray elements at the end of our computation. We take these constraints into account in our auto-tuning infrastructure, which runs our three vectorized tiled implementations using all legal parameter configurations while limiting the tiling factor and block size such that $B < L1 < 400$. This yields a `.csv` file containing the best configurations for each tested n. This file is then used as a input configuration for all subsequent benchmarks to guarantee a maximal performance.

Program Generation. While auto-tuning allows us to find the optimal parameters for the system we are working with, we still have one system-specific optimization which we can automate to achieve a high single-core performance. The idea is to maximize the ILP in our program by generating multiple implementations containing different amounts of unrolling and scalar replacement in the innermost loop. The goal with this is to find the optimal number of accumulators to use in order to maximize the ILP on the specific system used for our bench-marks. This technique is called program generation and is inspired by the method presented in a previous work on the subject [6].

The program generator is comprised of two components: the generator, and the tuner. The generator takes as input an unrolling factor and outputs a vectorized tiled implementation unrolling the inner-most loop by the given factor. The generated code is in Single Static Assignment (SSA) form,

meaning that each variable is assigned only once, allowing for the removal of all Write-After-Write (WAW) and Write-After-Read (WAR) constraints in the implementation. This will improve the overall ILP of the program. In order to achieve a good execution unit utilization, the generated vector intrinsics are grouped into load, compute, and store phases, thus improving the overall instruction pipeline.

The generator is invoked by the tuner, which compares the performance of different generated implementations and retains the unrolling factor that yields the best result. The program generator, along with the auto-tuner, allow for our solution to adaptive and extract the maximum amount of performance out of each system.

4. EVALUATION

We will now evaluate our fast implementation of the Floyd-Warshall algorithm. This is done using the three implemented semi-rings, and each optimization presented in section 3 will be used to obtain intermediary results. In the following paragraphs, we present the system on which we will conduct our bench-marking. After that, we show the different results obtained by comparing each level of optimization to each other. Finally we conduct a roofline analysis of our highest-performing implementations for each semi-ring to evaluate the quality of our optimizations.

Experimental Setup. We evaluate our implementations using an `Intel Coffee Lake` processor, which is part of the `Intel Skylake` architectural family. Table 1 summarizes the specifications of the processor used for the evaluation [14]. We compile our implementations using `gcc` version 9.4.0. We tested our implementations using many different combinations of compiler flags and found that the best results are obtained using the following flags [15]:

	Processor
Model	Intel® Core™ i7-9700K CPU
µArch	Coffee Lake (Skylake family)
Base Frequency	3.60 GHz
Max Frequency	4.90 GHz
Turbo Boost?	Yes
Tick-Tock-Opt	Opt
Total Cache Size	12 MB

Table 1. Specifications of the processor used for the evaluation.

- `-march=native`, which enables the use of architecture specific optimizations such as vectorization.
- `-O3`, which enables all basic optimizations, such as pre-computation, basic scalar replacement, loop splitting, in-lining, and many more.
- `-ffast-math`, which assumes floating point association enabling many other optimizations that allow for greater extraction of ILP.

Input Parameters. In our bench-marking infrastructure, we generate the inputs to our code using the same matrix structure presented in section 2. Thus for the {`OR, AND`} semi-ring we use `uint64_t` matrices stored in row-major order, while for the two other semi-rings, we use `double` matrices. For the tiled implementations, we use as input the parameters obtained through our auto-tuning process. Table 2 summarized the parameters which were used.

Semi-Ring	N	L1	B
{MIN, +}	8	8	8
{MIN, +}	24	24	24
{MIN, +}	72	72	72
{MIN, +}	216	72	72
{MIN, +}	648	108	108
{MIN, +}	1944	108	108
{MAX, MIN}	8	8	8
{MAX, MIN}	24	24	24
{MAX, MIN}	72	72	72
{MAX, MIN}	216	216	216
{MAX, MIN}	648	72	72
{MAX, MIN}	1944	72	72
{OR, AND}	8	8	8
{OR, AND}	24	24	24
{OR, AND}	72	72	72
{OR, AND}	216	108	108
{OR, AND}	648	72	72
{OR, AND}	1944	72	72

Table 2. Auto-Tuned parameters used as input for our tiled implementations.

Note that in every the optimal configuration has $L1 = B$. This shows that our tiling enables sufficient locality exploitation, and additional blocking yields negligible improvements. Additionally, we notice that for smaller sizes of n, tiling also yields negligible performance improvements, and thus the auto-tuner selected values such that $n = L1 = B$ when n is small. For n our main constraint, apart from avoiding two-powered strides, is that we need to guarantee the alignment of our matrices in order to avoid illegal memory accesses. This is done by starting with $n = 8$ and then maintaining multiples of 8 by simply increasing n at each iteration by a factor of 3.

Unrolling Factor for Program Generation. On top of requiring auto-tuned parameters, one of our implementations depends on an additional parameter called the unrolling factor. This factor affects the number of accumulators used in the generated implementation. Theoretically, the optimal number of accumulators per instruction can be

calculated using the gap, meaning the cycles per issue, by dividing the latency of the instruction by said gap, i.e. `n_acc = latency / gap` [16]. With the instruction mix defined by our three semi-rings, we obtain the same theoretically optimal number of accumulators, and thus in our case the unrolling factor, for all three implementations. Specifically, given that `latency = 4` and `gap = 0.5` for the {`MIN, +`} and {`MAX, MIN`} vector intrinsics, and `latency = 1` and `gap = 0.33` for the {`OR, AND`} ones, we get the following optimal unrolling factors:

- {`MIN, +`} and {`MAX, MIN`}: $\texttt{n\_acc} = \frac{4}{0.5} = 8$.
- {`OR, AND`}: $\texttt{n\_acc} = \frac{1}{0.33} = 3$.

When running our tuner on the generated implementations, we find that our empirical results match the theoretical ones, meaning that our empirically optimal unrolling factors are 8 for {`MIN, +`} and {`MAX, MIN`}, and 3 for {`OR, AND`}. These are thus the factors which we use when bench-marking the generated implementations.

Theoretical Peak performance. Before we present our results, we must define what the theoretical peak performance of our system is. We can compute the theoretical peak performance for computations in different semi-rings using the information present in the Intel Intrinsics Guide [17]. In all three computations, half of the operations (ignoring loads and stores) are semi-ring addition operations and the other half semi-ring multiplication operations. For {`MIN, +`} and {`MAX, MIN`}, all operations have a throughput of 2 vector operations per cycle. Since we are using vectors of four 64 bit doubles, the maximum performance is of 8 flops/cycle. In the case of {`OR, AND`}, both operations have a throughput of 3 vector operations per cycle. Here, we use vectors of four 64 bits integers yielding a theoretical maximum performance of 12 ops/cycle. This information is useful for better understanding the context of our performance plots.

Results. We now present the results obtained from our benchmarks. The goal of our experiments is to analyze the impact that each presented optimization has on the performance of our program in comparison to the baseline. To do this, we run each version of our program on the same system using the parameters from Table 2 as a configuration.

Figure 4 shows the different results achieved for each implementation on each evaluated semi-ring. For each semi-ring, we evaluate four different implementations:

- **Baseline**, presented in the first paragraph of section 3, does not contain any optimizations.
- **Tiled**, presented in the second paragraph of section 3, contains tiling and blocking optimizations, but no vectorization.

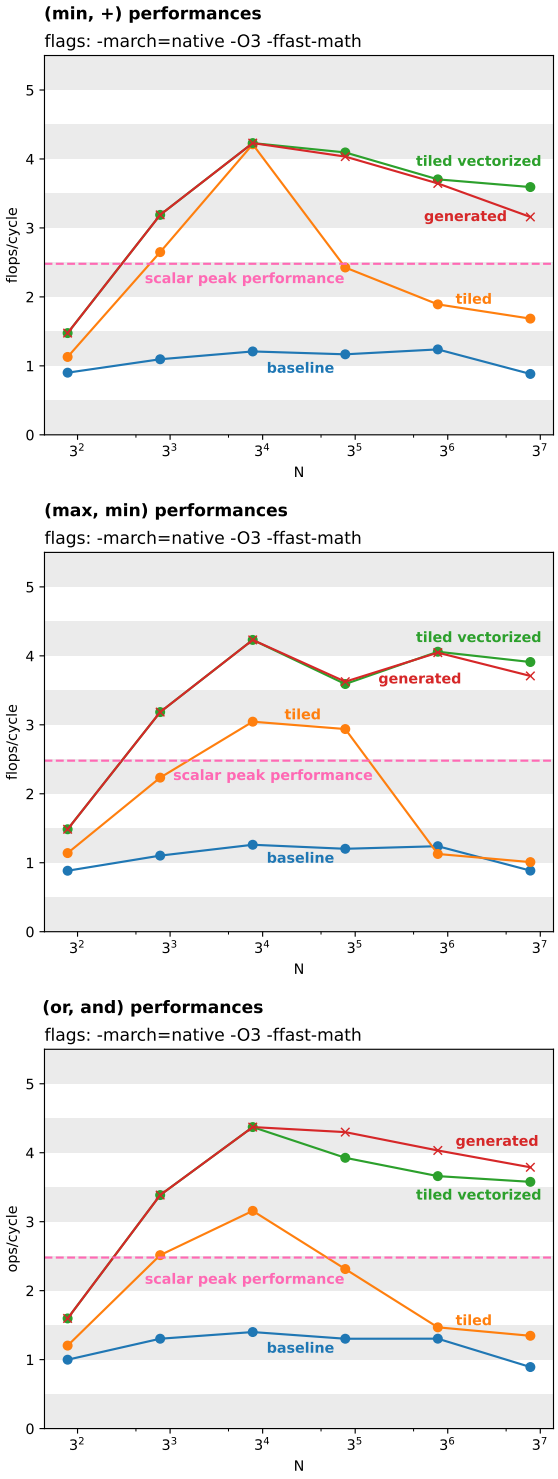

Fig. 4. Performance plots for the different implementations on the {`MIN, +`}, {`MAX, MIN`}, and {`OR, AND`} semi-rings. All of the implementations are compiled with the same flags. The scalar peak performance is empirically obtained through an analysis of our workload using Intel Advisor [18].

- **Tiled Vectorized**, presented in the third paragraph of section 3, adds explicit vectorization, using vector intrinsics, to the tiled implementation.

- **Generated**, presented in the last paragraph of section 3, uses the program generator to add the optimal amount of scalar replacement and unrolling to the implementation.

A first observation is that the tiled implementation, which is not explicitly vectorized, still manages to surpass the empirical scalar peak performance. This is due to the the compilation flags used, which enable a form of automatic vectorization. However, as is observable in the results, the compiler does not manage to vectorize the code better than an explicit vector implementation, and thus the tiled implementations performs worse than the tiled and vectorized one. We observe as well that all three semi-rings behave very similarly in terms of performance growth. The baseline remains quite stable while the tiled implementations grow as n grows until $n = 72$, after which the vectorized implementation maintains a stable performance, but the non-vectorized implementation suffers a strong performance drop. Since our tiled implementations manages to exploit data locality in the cache as much as possible, we hypothesize that this drop is due to the compiler no longer being able to auto-vectorize the implementation for large values of n. A final observation is that the generated code seems to only perform slightly better than our tiled and vectorized implementation. This is expected since it only adds a slight improvement in the overall ILP. We expect that the generated code would continue to perform better than our tiled and vectorized implementation on very large values of n.

Table 3 shows the average speedups obtained between each implementation for all three semi-rings. The speedups are computed by taking the performance of each implementation, given in flops/cycle, and dividing it by the performance of the implementation we are comparing it to. In the table itself, the columns are compared to the rows. We can see that in all three cases, the generated code performs very similarly as the vectorized implementation, for the reasons present earlier in the section. In comparison to the baseline however, we obtain a speedup of around $3\times$ for all three semi-rings. Analyzing the speedup between the different implementations shows that tiling contributes to our performance gains as much as explicit vectorization, since our non-vectorized tiled implementation already boasts an average $2\times$ speedup over the baseline for all three semi-rings. Finally, we can also see that the overall speedup for the $\{$`OR`, `AND`$\}$ semi-ring is less than for the two other ones. This can be attributed to the different data types being operated on, which will inherently yield different results, given that the latencies and gaps of the operations used are fairly different.

Existing Fast Implementation. In order to better understand the quality of our optimizations, we compare our generated $\{$`MIN, +`$\}$ implementation with an existing fast implementation of the Floyd-Warshall algorithm. The only open-source, and functioning, implementation we have found is a `numpy` implementation of the algorithm [19]. All other implementations available are either broken or slower than this one. Figure 5 shows the results from our comparison

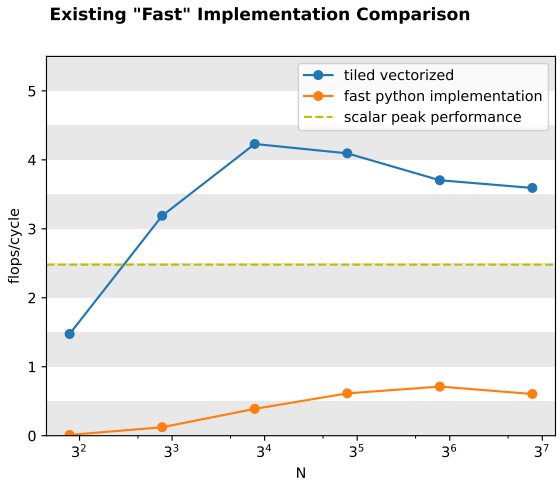

Fig. 5. Performance plot comparing the fastest available single-core open-source implementation to our generated tiled implementation.

between the aforementioned existing fast implementation and our generated one. This clearly shows that our implementation far the existing open-source one, yielding an average $7\times$ speedup in comparison to the `numpy` implementation. Note that this result is expected given that `python` implementations are notoriously slow. However, this is the only available implementation which attempted any sort of optimizations, which is why we chose to include it in our analysis.

Roofline Analysis. A final important analysis to do in order to contextualize the performance within the scope of our system is to express our results within a roofline model of our system [20]. In order to obtain the necessary data to compute the operational intensity of our computations, we use the `perf` performance counters. These allow us to measure the number of cache misses as well as the miss rate itself for our generated implementations. Using this, we find that our implementation almost perfectly optimizes for cache usage, since we have a miss rate that drops as low as 0.02% for some values of n. However this seems to be greatly dependent on the size of the matrix used. This information is then used to compute the operational intensity of

MP	B	T	V	G
B	1.0	0.53	0.35	0.347
T	2.11	1.00	0.69	0.69
V	3.10	1.54	1.0	0.99
G	3.10	1.55	1.003	1.00

MM	B	T	V	G
B	1.00	0.68	0.35	0.36
T	1.70	1.00	0.59	0.60
V	3.04	2.14	1.00	1.02
G	3.02	2.08	0.98	1.00

OA	B	T	V	G
B	1.00	0.65	0.37	0.38
T	1.63	1.00	0.59	0.61
V	2.86	1.81	1.00	0.99
G	2.89	1.86	1.01	1.00

Table 3. Summary of the different speedups obtained on the different versions of our program working on the three different semi-rings. The semi-rings are presented in the following order from left to right: {`MIN`, `+`} (MP), followed by {`MAX`, `MIN`} (MM), and finally {`OR`, `AND`} (OA). The compared versions are the baseline (B), the tiled version (T), the tiled and vectorized version (V), and the generated version (G). The speedup is computed as the ratio between the performance of the column implementation and the performance of the row implementation.

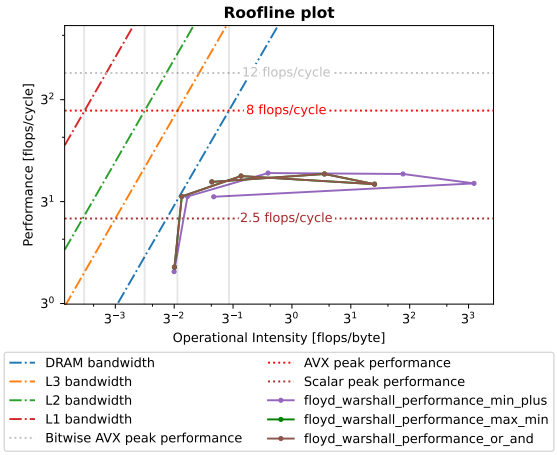

Fig. 6. Roofline plot of the three generated implementations running on our bench-marking system. The diagonal lines on the left represent the different memory bandwidths within our system. The bandwidths are the following (with lines ordered from left to right): L1 is 386.55 GB/s, L2 is 125.1GB/s, LLC is 67.8 GB/s, and Main Memory is 25.91 GB/s.

our computations using the following formula:

$$I(n) = \frac{\#\text{ops}}{\#\text{cache\_misses} \times \text{block\_size}}$$

This information, along with the performance results from our benchmarks, allow us to create our roofline model. Figure 6 plots the results of our generated implementations within a roofline model of our system. We can observe that all three semi-rings have a very similar performance behavior, and reach within $2\times$ the maximal theoretical performance given the instruction mix. Our benchmarks, however, are not run in a perfectly isolated environment and thus such a perfect performance is unlikely to be achievable. Analyzing the roofline plot, we can observe that our computation begins as a memory-bound problem, meaning that the memory bandwidth is the main bottleneck [21], and then shifts towards being computationally-bound once $n > 72$. We can also observe that our operational intensity grows along with n until it reaches $n > 618$. At this point, the operational intensity drops drastically. We hypothesize that this is due to the matrix no longer fitting in cache, since at $n = 1944$ we have $1944^2 = 3779136$ 64 bit elements per matrix, which would take up $3779136 \times 8 = 30233088\text{B} \approx 28.83\text{MB}$, which is superior to our cache size of 12MB. This would thus lead to a large increase in cache misses, and thus a lower operational intensity.

5. CONCLUSION

Throughout this work, we explored the different optimization routes that can be taken to improve the performance of the Floyd-Warshall algorithm using only hardware that is available on a single core. Doing, so we found that the best optimizations for this problem were matrix tiling, which improved the exploitation of both spatial and temporal data locality, and explicit AVX-2 vectorization, which use vector intrinsics to manually exploit the super-scalar hardware available on most modern processors.

Our goal was also to provide an adaptive solution, which would optimally exploit the hardware of any system it could be run on. This was achieved through auto-tuning and program generation. Auto-tuning yielded very promising speedups over the worst parameter choice, and thus should always be used over manually selecting parameters. However, program generation gave us almost disappointing results, since it almost always performed very similarly to our tiled and vectorized implementation. In the future, this could be improved by implementing smarter parameter search heuristics, rather than brute-force trying all possibilities, since this would allow us to try a larger range of parameters and generate code for larger matrix sizes.

Our final results showed that used these optimization approaches, we were able to obtain a $3\times$ speedup over the baseline for all three semi-rings. These results were within $2\times$ of the theoretical max-performance for our instruction mix. The main takeaway from this work is that locality optimizations and vectorization are indispensable if the highest possible performance is to be achieved on a single core.

6. CONTRIBUTIONS OF TEAM MEMBERS

Contributions were very homogeneous among the group. We held weekly meetings to detect issues and work on them together as a group. Therefore, a list specifying all the contributions of each member is bound to be incomplete. The individual contributions of each team member can be summarized as follows:

Andrew. Focused on tiling and blocking for {`MIN, +`} and {`MIN, MAX`}, as well as the auto-tuning infrastructure. Also worked on program generation (with Shashank), and on the roofline analysis (with Michele).

Marie. Worked on Vectorization, as well as the bench-marking and plotting infrastructure. Also worked on program generation (with Shashank).

Michele. Focused on everything related to the {`OR, AND`} semi-ring. Also worked on the roofline plots.

Shashank. Focused mainly on program generation, as well as vectorization, creating the bench-marking infrastructure, and comparison with the fast existing implementation. Worked on tiling and blocking (with Andrew).

7. REFERENCES

[1] Vaishnavi Sonwalkar, "Floyd warshall algorithm: applications," https://vsonwalkar3.medium.com/floyd-warshall-algorithm-8ac60b903b14, 2020.

[2] Ivan Jovanovic, "What are the applications of the shortest-path-algorithm?," https://stackoverflow.com/questions/4412031/what-are-the-applications-of-the-shortest-path-%20algorithm, 2010.

[3] Ben D. Lund and Justin W. Smith, "A multi-stage cuda kernel for floyd-warshall," *ArXiv*, vol. abs/1001.4108, 2010.

[4] Li-yan Zhang, Ma Jian, and Ke-ping Li, "A parallel floyd-warshall algorithm based on tbb," in *2010 2nd IEEE International Conference on Information Management and Engineering*, 2010, pp. 429–433.

[5] Szymon Pluta, "Fast floyd-warshall," https://github.com/Wenox/fast-fw, 2021.

[6] 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.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Third Edition*, The MIT Press, 3rd edition, 2009.

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

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

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

[11] Markus Püschel, "ASL Lecture: Dense linear algebra, lapack/blas, atlas, fast mmm," https://acl.inf.ethz.ch/teaching/fastcode/2022/slides/10-linear-algebra-MMM.pdf, Given: 2022-03-31.

[12] Markus Püschel, "ASL Lecture: Memory hierarchy, locality, caches," https://acl.inf.ethz.ch/teaching/fastcode/2022/slides/08-locality-caches.pdf, Given: 2022-03-21.

[13] Yoongu Kim and Onur Mutlu, "Memory systems," in *Computing Handbook, Third Edition: Computer Science and Software Engineering*, Teofilo F. Gonzalez, Jorge Diaz-Herrera, and Allen Tucker, Eds., pp. 18: 1–21. CRC Press, 2014.

[14] "Intel® core™ i7-9700k processor," https://www.intel.com/content/www/us/en/products/sku/186604/intel-core-i79700k-processor-12m-cache-up-to-4-90-ghz/specifications.html.

[15] "Options that control optimization," https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[16] Markus Püschel, "ASL Lecture: Optimization for instruction-level parallelism," https://acl.inf.ethz.ch/teaching/fastcode/2022/slides/04-ILP.pdf, Given: 2022-03-03.

[17] "Intel intrinsics guide," https://www.intel.com/content/www/us/en/docs/intrinsics-guide/ .

[18] Diogo Marques, Helder Duarte, Aleksandar Ilic, Leonel Sousa, Roman Belenov, Philippe Thierry, and Zakhar A. Matveev, "Performance analysis with cache-aware roofline model in intel advisor," in *2017 International Conference on High Performance Computing & Simulation (HPCS)*, 2017, pp. 898–907.

[19] Mattia Neroni, "Fast floyd-warshall," https://github.com/mattianeroni/Fast-Floyd-Warshall, 2021.

[20] 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.

[21] Markus Püschel, "ASL Lecture: Roofline model," https://acl.inf.ethz.ch/teaching/fastcode/2022/slides/09-roofline.pdf, Given: 2022-03-28.