On the compilation of a parallel language targeting the self-adaptive virtual processor

Bernard, T.A.M.

Citation for published version (APA):
Chapter 7

SVP evaluation

Compiler development requires, as with any complex software development, special care with regard to code quality. A compiler can be thought of as a gas factory where an improvement to its infrastructure consists of adding or even removing pipes. After this operation, the gas factory must be checked for leakage in case changes introduced some inconsistencies. Any extension in a complex software system suffers from consistency issues; the entire software system needs to be certified before it is safe to use. In the context of this research, we endorsed as far as possible compiler correctness. Adding code, adding components, removing segments from the compiler infrastructure may render it unstable and become non-certified if compiler architects do not pay particular attention to any side-effects. In this regard, Test-Driven Development (TDD) \[89\] enables, in short development cycles, the verification of changes made in the compiler internals. TDD results in using specific test cases and scenarios to trigger problems in compiler behavior or compiler produced code. A few lines of changes can then be verified quickly and verify the range of side-effects they may impact on other compiler components. Chapter 5 has discussed how SVP properties endanger conventional compiler optimizations if they are not extended. Therefore, achieving compiler correctness and correctness of produced code are big challenges in our research. This presence of a working compiler for SVP is already a proof of concept. This chapter thus talks about code analysis, after Chapter 6 has examined compiler conception and decisions.

The contents of this chapter are based on this publication:

Chapter 7. SVP evaluation

Chapter 3 has outlined evaluation of the SVP software system including scientific problems expressed in highly-optimized hand-coded benchmarks for the SVP architecture implementation. The first part of this chapter reviews the SVP extension of the compiler. Evaluation gathers code analysis of scientific problems: code size, code composition, code correctness and compiler overhead. The idea is to monitor impact and overhead of the SVP extension onto the imperative-language compiler. We then analyze SVP produced code to reveal composition of programs and distribution of SVP and conventional operations. Finally, performance differences between hand-coded (in the Microgrid ISA) programs and compiled programs give an estimate of our research compiler in finding an optimal solution when compiling a problem. The second part of this chapter contrasts with the results section in Chapter 3 which has used as benchmarks hand-coded programs; the second part utilizes compiled programs as benchmarks. We provide an evaluation (with results) of the SVP parallel computing system comprising an SVP hardware implementation as the Microgrid, an SVP software implementation as the µTC language, an SVP GCC-based compiler implementation and a set of scientific problems programmed in µTC.

7.1 Evaluation of SVP compilation

This first section focuses on the SVP research compiler and evaluation of SVP compilation. Compiler correctness is our main objective. Moreover, code analysis of scientific problems shows how the SVP compilation performs: composition of SVP compiled programs (instruction mix), the importance of SVP-aware optimizations in SVP compilation.

7.1.1 Methodology: benchmarks and evaluation

The Livermore kernels are characterizations of scientific problems used for benchmarking in this chapter. In our compiler research, they correspond to an acceptable trade-off between program size and problem complexity. Their program size allows us to monitor each operation in compiled code with a human eye, whereas larger programs would complicate the monitoring with huge numbers of operations. Their problem complexity is a collection of common scientific mathematical problems, where some of them can be parallelized easily and others require more attention. They test code correctness and evaluate the code quality of the SVP compiler (µTC input language). Our SVP compiler is an extension of a conventional imperative-language compiler. It is based on the GCC-4.1 framework and extended by the presented compilation schemes from the µTC language to Alpha-based SVP cores (cf. Chapter 4), for which we have a many-core (Microgrid) emulator which provides a cycle-accurate simulation of execution time. The Livermore kernels gather a set of scientific kernels which have been coded in µTC: Hydrodynamics Fragment (HF); Inner Prod-
uct (IP); Banded Linear Equations (BLE); First Differential (FP); First Sum (FS); Incomplete Cholesky Conjugate Gradient (ICCG); Difference Predictors, (DP); Tridiagonal Elimination (TDE); General Linear Recurrence Equations (GLRE); Equation of State Fragment (ESF). Our toolchain allows verifications of compilation results and execution on this platform against the execution of the same code on a conventional platform by transforming $\mu$TC into sequential C code, compiling it and executing it on a conventional processor.

7.1.2 Code analysis and quality

Figure 7.1 presents the distribution of operations over the entire set of kernels in their assembly representation (i.e. Microgrid ISA). Across scientific problems, the distribution of extra SVP operations goes from 14% for ESF to 37% for FS - ratio of SVP operation overall the program. ESF has more arithmetic and logical operations (ALU ops) than FS. The number of SVP operations is therefore not negligible; these operations are responsible for thread creation (create in Figure 7.1) and inter-thread communication in shared channels via shared-dependent operations (S/D ops). There are also directives for register window definition and global data pointer (directives), context switching tags (swch) - note that these are not considered as operations but still shown in Figure 7.1).

Further code analysis is shown in Figure 7.2 where unoptimized (GCC -O0) and optimized (GCC -O1) $\mu$TC are compared. The unoptimized code has a bigger code size than the optimized version, because the GCC compiler does not use any code optimization and uses the stack for parameter passing and for storing intermediate results. Optimized versions prioritize the use of reg-
isters before the stack for parameter passing; code size drastically diminishes, by almost 30% in all programs. Moreover, to achieve this gain, major compiler optimizations needed to be extended to support the SVP assumptions with the -O1 optimization flag, as reported in Chapter 5.

**Figure 7.2**: Comparison of code size between unoptimized and optimized µTC code. Unoptimized code is generated with the debugging -O0 optimization flag of GCC, whereas optimized code uses optimization flag -O1. These results are generated with the SVP GCC-4.1 compiler. The y-axis represents the code size (octets).

Figure 7.3 collects a comparison between hand-coded implementations versus their corresponding representations in µTC. The focus here is to monitor how close the compiler is to an optimal solution. Hand-coded versions are extensively optimized by SVP hardware specialists; therefore, hand-coded programs benefit from every single Microgrid property. In some cases, the compiled code is much larger than the hand-coded version, e.g. HF is 204% larger in size than its hand-coded version. The worst case is for IP where the compiled version is 285% bigger. A possible criticism to the SVP compiler is the unique use of the -O1 optimization flag. The other optimization flags such -O2 and -O3 might give better results; however, the impact of SVP properties on compiler optimizations has not been investigated and will be left for further research. The SVP compiler is unstable with this aggressive optimization flag. Experiments with optimization flag -O2 mainly result in clashes with the instruction rescheduling pass of the compiler. Further work is required to take advantage of these optimization passes. It should be noted that the GCC compiler community struggles with the most aggressive (-O3) when combined with code (this can be observed with the amount of active discussion on the GCC developer mailing list).

### 7.1.3 Code correctness

Figure 7.4 compares the execution time of compiled Livermore kernels against optimized hand-coded ones. These compiled execution times are normalized
Chapter 7. SVP evaluation

Figure 7.3: Comparison of instruction/directive size between hand-coded and compiled μTC code (μTC implementation of Livermore kernels). Hand-coded programs correspond to an optimal solution of SVP compilation. μTC code is compiled with optimization flag -O1 with the SVP compiler. The bar chart display an analysis of instruction/directive size for each scientific problem. These results are generated with the SVP GCC-4.1 compiler. The y-axis represents the number of instructions.

against the hand-coded versions for execution on a single core and a cluster of 32 cores. The code correctness is certified by the proper execution of the kernels on the target platform (Microgrid) simulator. We evaluate the compiled code against the hand-coded assembly versions that can be considered as optimal solutions. We would expect the compiled code to be slower than the highly-optimized hand-coded assembly. We would also expect the code to scale with the number of cores, and here we see the compiled μTC code scaling much better than the assembly version. For a single core, the compiled code varies from being as good as the hand-coded version (BLE) to a factor of 2.8 times slower (IP). IP performs only one multiplication and one addition in each thread and less efficient address arithmetic can not be masked. The average slowdown is just about 50%. When we execute on 32 cores, however, the smaller hand-compiled kernels execute less efficiently, as there are fewer threads per core and more stalls waiting for memory or synchronizations (e.g. in IP), whereas the longer kernels continue to utilize the pipeline more efficiency. At 32 cores the overhead of compiled code becomes less than 20% and in few cases, HF, IP, BLE and ESF are very close to the hand-coded assembly version.
Chapter 7. SVP evaluation

7.2 Evaluation of SVP computing system

This second section collects an evaluation of the SVP computing system comprising a target platform (i.e. the Microgrid simulator in Section 7.2.1) benchmarks (cf. Section 7.2.2) compiled with our research SVP compiler. A major aspect of this evaluation is the reuse of the same program’s binaries across different settings of the chip multi-processors (i.e. Microgrid) without need of recompilation: number of cores, problem size, etc. This resource agnosticism is a property of the SVP execution model. This section provides an overview of the performance of the SVP computing system.

7.2.1 Target platform: Microgrid simulator

The SVP software simulator [68] keeps track of the full state of the system, simulates the pipeline, caches, network and memory and takes contention on those buffers and ports into account to provide a cycle-accurate simulation of execution time. The simulated system consists of 128 1.2 GHz cores spread among several differently-sized places. Each core has a 4-way set associative 1 kB I-Cache and D-Cache, storage for 32 families, 256 threads and 1024 integer and 512 FP registers. A 4-way set associative 32 kB L2 cache serves 4 cores via a bus and a COMA directory is connected with 8 L2 caches in a ring. Two DDR3-2400 channels are connected to the top-level directory ring.

The evaluation, performed in this section, uses this platform emulation of a Microgrid. The emulation captures state transitions down to the lowest level
in the cores’ pipelines. It also provides a fully parameterizable, cycle-accurate simulation of all hardware aspects of the Microgrid: core functional units, interconnects, network-on-chip, memory architecture, and external memory interface. The memory architecture is a COMA derivative described in [70]. We used parameters suitable for hardware that can be built using current technology [61, 65]: 128 SVP cores implementing the DEC Alpha ISA, sharing 64 FPUs with separate pipelines for adds (2 pipelines), divs, muls and square root operations (1 pipeline each); 32 L2 caches of 32 kB each shared by group of 4 cores. L2 caches are in turn grouped in COMA rings of 8 caches connected to a COMA directory. The top level directory ring is connected via two DDR3-2400 channels to external storage of arbitrary size. The performance figures presented in the next section should be considered in the light of the following hardware characteristics: the two DDR3-2400 channels provide 2400 million 64-bit transfers/s of 64 bits each, i.e. a peak bandwidth of 38.4 GB/s overall for the external memory interface; each COMA ring provides a total bandwidth of 64 GB/s, shared among its participants; the bus between cores and L2 caches provides 64 GB/s of bandwidth, shared among 4 cores; the SVP cores are clocked at 1.2 GHz. The cores are grouped in 8 clusters (SVP places) of 1, 1, 2, 4, 8, 16, 32 and 64 cores, to allow running benchmarks on places of varying sizes. The selection of cluster sizes has no impact on the performance of each cluster other than the number of cores. Figure 7.5 provides a schema of this configuration.

7.2.2 Methodology: benchmarks and evaluation

We present results spanning a selection of computation kernels present in most applications. The selection was made among known scientific problems (Livermore kernels, linear algebra, etc.) as they are commonly used in benchmarks. Our selection process was guided by the intent to expose the relationship between implementations, architecture parameters and actual observed performance.

Figure 7.5: Functional diagram of a 64-core Microgrid.
Most results are presented with performance on cold caches (first use) in addition to the more common warm caches performance. This allows us to quantify the impact of kernels that may be used only once in a larger computation. In all figures, the problem size indicates the number of items on which the kernel computation is applied. This closely matches the number of SVP threads defined.

In order to analyze the performance, we need to understand the constraints on performance. For this we define two measures of arithmetic intensity (AI). The first $AI_1$ is the ratio of floating point operations to instructions issued. For a given kernel that is not I/O bound, this limits the FP performance. For $P$ cores at 1.2 GHz, the peak performance we can expect therefore is $P \times AI_1$, the ideal case of full pipeline utilization (one apparent cycle per operation). In some circumstances, we know that execution is constrained by dependencies between floating point operations, so we modify $AI_1$ to take this into account giving an effective intensity $AI'_1$. The second measure of arithmetic intensity is the ratio of FP operations to I/O operations, $AI_2$ FLOPs/byte. I/O bandwidth $IO$ is usually measured at the chip boundary (38.4 GB/s - the two DDR3-2400 channels provide 2400 million 64-bit transfers/s, i.e. $2400 \times 2 \times 8 = 38.4 \text{GB/s}$), unless we can identify bottlenecks on the COMA rings (64 GB/s). These I/O bandwidths are independent of the number of cores used, so it also provides a hard performance limit. We can then combine these two intensities to obtain a maximum performance envelope for a given code and problem size. A program is either constrained by $AI_1$ if $P \times AI_1 \leq AI_2 \times IO$ or $AI_2$ when $P \times AI_1 \geq AI_2 \times IO$.

### 7.2.3 Sequential code

We consider the function DNRM2 of the BLAS library. This function computes the Euclidean norm of a vector: it sums up the squares of the vector elements, then computes the square root of the sum. In the sequential algorithm, each iteration requires one memory load, followed by an FP multiply-add. The add in each iteration has a data dependency on the previous iteration. We do not yet consider here the benefits of implementing this function using another algorithm based on e.g. a parallel reduction. This will be evaluated in Section 7.2.4.

Importantly, one should consider that any architecture must deal with memory latencies of hundreds of cycles or more in scheduling instructions. Branch prediction and out-of-order issue can provide some latency tolerance, typically tens of cycles, which is sufficient to optimize performance when working from on-chip cache but not for larger data sets. Prefetching can do better on constant-stride accesses but as memory latencies rise, the probability that prefetched data will remain in the cache diminishes. In our approach, the hardware provides latency hiding through multithreading in the pipeline: the load and FP mul form an independent prefix to the dependent add, so latency hiding is possible by overlapping them across threads and across cores. In principle, we can expect that the latency of loads can be hidden even with cold caches, and that the resulting performance is the sequential performance of the dependent FP
operations assuming no delays.

```cpp
thread accum(shared double sum, double* v) {
    index i;
    sum += v[i] * v[i];
}
thread dnrm2(shared double result, int n, double *v) {
    create(fid; ; 0; n; 1;) accum(sum = 0, v);
    sync(fid);
    result = sqrt(sum);
}
```

Figure 7.6: BLAS DNRM2 in \( \mu TC \).

We have a naive implementation of DNRM2 in \( \mu TC \) (cf. Figure 7.6). This compiles to code with 10 instructions per thread, including 3 FP operations. In each thread, the 7 initial instructions are entirely independent and can run concurrently. So, latency hiding applies and we can expect a visible 1-cycle cost per instruction. Then, the dependent adds are executed in sequence. The cost of this is between 6 and 11 cycles per add depending on the scheduling of threads (considering a 6-stage pipeline and chip layout), with the difference representing the cost of waking up a waiting thread and getting it to the register read stage of the pipeline, which could be overlapped by other independent instructions in the pipeline. To this we add the cost of the 7 independent instructions yielding 6 + 7 to 11 + 7 cycles per thread. Even overlapping adds with the independent instructions, we can not hide the minimum 13 cycles latency from add to add as they are dependent. We then deduce the arithmetic intensity \( AI_1 = \frac{\text{nbr of FP operations}}{\text{nbr of instructions}} \), i.e. \( AI_1 = \frac{3}{10} = 0.3 \). We then define the effective intensity, \( AI'_1 \) considering the dependencies between adds, nbr of FP ops \( \frac{\text{nbr of instructions} + \text{cost}_{\text{max}}}{\text{nbr of instructions} + \text{cost}_{\text{min}}} \) ≤ \( AI'_1 \) ≤ \( \frac{\text{nbr of FP ops}}{\text{nbr of instructions} + \text{cost}_{\text{min}}} \), i.e. \( (3 \div (10 + 18)) \approx 0.10 \leq AI'_1 \leq 0.13 \approx (3 \div (10 + 13)) \). Deriving from \( AI'_1 \), it gives a theoretical maximum performance envelope - for 1.2 GHz cores - of 0.11 to 0.15 GFLOP/s on a single core.

As Figure 7.7 shows, provided we have enough threads we observe just under 0.11 GFLOP/s on one core, i.e. within the expected range. We have confirmed experimentally that this sort of gain can be reproduced for a variety of memory architecture configurations with diverse delays with the only difference being the number of threads required to tolerate the different memory latencies.
Chapter 7. SVP evaluation

(a) Performance on cold caches

(b) Performance on warm caches

Figure 7.7: Performance of DNRM2 on one SVP place. The working set is $8 \times \text{#problem size}$ bytes.
Chapter 7. SVP evaluation

We observe that we can further increase the performance by increasing the number of cores, because the independent prefix instructions can be scheduled concurrently across cores. However, not much gain is to be expected since less than one third of the cycles required can be executed concurrently. Even with ideal scheduling and no overhead Amdahl’s law would limit speedup to a factor 1.5. Any deviation from this expected performance is due to insufficient threads to hide the asynchronous operations such as those to memory (e.g. problem size N=2K with cold caches in Figure 7.7(a)). Note that with warm caches less threads are required to tolerate memory latencies which are now smaller. We nonetheless observe with warm caches (e.g. size N=50K at 32 cores in Figure 7.7(b)) the performance drops beyond a problem size of N=50K. At that point, caches are full and new cache lines are required for computations - i.e. cache eviction. On a larger problem size, the overall performance gets better since the number of threads provides enough latency hiding.

We conclude this subsection by highlighting that we have achieved the automatic, fine-grained parallelization of the concurrent part of a purely sequential algorithm. This result is significant considering that our software implementation is naive, resource-agnostic and that the hardware scheduler is not specifically optimized for this algorithm.

7.2.4 Parallel reductions

The case study in this section is the inner vector product (IP, loop 3). For this algorithm, we have assumed associativity of the floating point addition used in the reduction as an opportunity to parallelize across cores. Our program (cf. Figure 7.8) is a straightforward extension of the naive implementation in \( \mu \)TC, which could be identified and transformed automatically. It relies on the number of cores in the ‘current place’ being exposed to programs as a language primitive. It splits the reduction into two stages, where an initial family is created with one thread per core, and each thread in that family then creates a local family to perform a core-local reduction. In effect, this amounts to parallelizing the reduction among the cores. So, when the number of threads per core is significantly larger than the number of cores, the cost of the final reduction is negligible and the performance should scale linearly with the number of cores. Figure 7.9 shows the experimental results for this code.

To analyze our results, we consider the case of one core first. The compiled code for the inner thread contains 7 instructions, including 2 FP operations and 2 loads. The arithmetic intensity is \( AI_1 = 2 \div 7 \approx 0.28 \). There are dependences between threads and instructions (i.e. dependent FP add); using the same methodology as before, we get a cost per thread of between 6 and 11 cycles. We deduce the effective intensity \( AI_1' \), i.e. \( (2 \div (7 + 11)) \approx 0.11 \leq AI_1' \leq 0.15 \approx (2 \div (7 + 6)) \). The effective intensity hence provides a theoretical maximal performance envelope on one core - for 1.2 GHz cores - of 0.13 to 0.18 GFLOP/s. The actual performance observed falls within this range (0.15 GFLOP/s). Assuming linear speedup on the number of cores, the expected performance for P
cores is thus \(0.15 \times P\) or a theoretical maximum of 9.6 GFLOP/s for the largest place size of 64 cores.

\begin{verbatim}
thread accum2(shared double sum, double* x, double *y) {
    index i; sum += x[i] * y[i];
}
thread accum1(shared double sum, int sp, double* x, double *y)
    {
        index i;
        create(fid; LOCAL; i; i+sp; 1; ) accum2(sum1 = 0, x, y);
        sync(fid);
        sum += sum1;
    }
thread lmk3(shared double result, int n, double *x, double *y)
    {
        int p = current_number_of_cores();
        create(fid; ; 0; p; 1; 1) accum1(sum = 0, n/p, x, y);
        sync(fid);
        result = sum;
    }
\end{verbatim}

**Figure 7.8**: \(N/P\) parallel reduction for the inner product.

However, the arithmetic intensity \(AI_2\) of this algorithm is just one FLOP per byte loaded from memory, i.e. \(AI_2 = 2\) FP ops \(\div 2\) I/O ops. By increasing the number of cores we eventually saturate the external memory bandwidth which - as described in Section 7.2.1 - is 38.4 GB/s. When the working set does not fit in the L2 caches yet another constraint is added. Loads to memory must now be interleaved with evictions from L2 cache when full. In the worst case, a single load may evict a cache line where the loaded line is used only by one thread before being evicted again. However, as threads are scheduled initially in index order this extreme case is unlikely and up to eight loads (64 Byte cache lines) may be serviced for one load-eviction pair. Evictions are either injected into another cache in the same ring or update the root directory without requiring off-chip bandwidth. The COMA ring bandwidth - as described in Section 7.2.1 - is 64 GB/s.
Figure 7.9: IP performance, using N/P reduction. The working set is $16 \times \#\text{problem size}$ bytes.
Thus, the ring bandwidth required for a single 8-byte load could be as much as two 64-byte cache-line transfers giving a perceived bandwidth of 4 GBytes per second, which with an arithmetic intensity of one eighth of a FLOP per byte gives a worst case expected performance to 0.5 GFLOP/s when the problem size is very much larger than the cache capacity, i.e with frequent evictions.

As our results show, with cold caches, for sufficiently large problems we achieve 6.3 GFLOP/s somewhat less than the theoretical maximum which is 9.6 GFLOP/s. This is due to the overheads of the sequential reduction phase. With warm caches, as we increase the problem size further, the program becomes constrained by eviction bandwidth. However, the number of threads provides enough latency hiding beyond N=50K.

We consider this ability of our resource-agnostic software implementation to be bound only by simple architectural constraints as another significant result and contribution of this paper.

### 7.2.5 Data-parallel algorithms

We show here the behavior of three data-parallel algorithms which exhibit three different behavior patterns. The equation of state fragment (ESF) from the Livermore benchmark suite (loop 7) is a data parallel kernel with seven local accesses to the same array data by different threads. If this locality can be exploited the kernel has an arithmetic intensity $AI_2$ of about 0.5 FLOPs/Byte of I/O from off-chip memory. Matrix-matrix product from the Livermore benchmark suite (loop 21), which although has significant non-local access to data, has a very high arithmetic intensity $AI_2$ as specified here, where one matrix can remain in on-chip memory giving an arithmetic intensity $AI_2$ of over 3 FLOPs per byte of I/O to off-chip memory. The 1-D FFT lies somewhere between these two extremes it has a logarithmic number of stages that can exploit reuse (linear for matrix multiplication) but has poor locality of access to data. As for our other benchmarks, our μTC implementations here are straightforward parallelizations of the obvious sequential implementation and do not attempt any explicit mapping to hardware resources.

We consider the ESF first. This takes 3 arrays and 3 scalars as input and computes an output array with the following expression:

$$X[k] = U[k] + R \ast ( Z[k] + R \ast Y[k] )$$
$$+ T \ast ( U[k+3] + R \ast ( U[k+2] + R \ast U[k+1] ) )$$
$$+ T \ast ( U[k+6] + Q \ast ( U[k+5] + Q \ast U[k+4] ) ) ;$$

The compiled code for the inner thread contains 33 instructions, including 9 loads, 1 store, 8 FP adds and 8 FP muls. As before, assuming latency hiding, each instruction should execute with a ‘visible cost’ of 1 cycle. The threads are independent; therefore, there is no need to determine the effective intensity $AI'_1$. The performance bound based on this mix of instructions should thus be 16 FLOPs in 33 instructions which gives an arithmetic intensity $AI_1$ of 0.49. We
derive the performance from it - considering 1.2 GHz cores - of 0.58 GFLOP/s. The observed performance on one single core is 0.49 GFLOP/s, i.e. \( \approx 85\% \) of the expected maximum (see Figure 7.10(a)).

As for the IP above, when there are enough threads per core to hide latency, we observe linear speedup on up to 8 cores. After this point, the program becomes I/O bound. For this program, an arithmetic intensity \( AI_2 \) of 0.5 FLOP-s/Byte assuming the eviction of the cache lines containing the results means a peak performance of 12.8 GFLOPS would be expected based on perfect sharing. Our experiments show that we reach 10.3 GFLOP/s for \( P=64 \) and \( N=1K \), i.e. a little over 80\% of this theoretical maximum. The bound may be loose because of constraints from internal cache traffic required to achieve this sharing. Again, as for the IP above when using larger problem sizes, L2 cache evictions cause the effective usable bandwidth to decrease and the performance drops accordingly. Conversely, when using warm caches and smaller problem sizes, greater speedups can be achieved (see Figure 7.10(b)). With warm caches, the program is computation-bound and the performance flattens after a size over \( N=2K \). The cache lines keep being evicted by new ones. The observed maximum - at \( P=64 \) cores, \( N=2K \) - is 22.9 GFLOP/s.

The next benchmark, in Figure 7.11, is a matrix-matrix product which implements naively the product of \( 25 \times 25 \) and \( 25 \times N \) matrices using a local IP algorithm. The IP operates in each core on rows of 25 consecutive elements in memory, thus achieving near-ideal locality of reference for the first matrix, which easily fits into on-chip cache. Even for large problems, the algorithm has a high arithmetic intensity as, for each column of the second matrix loaded, 1250 FLOPs are required to produce a column of the result (25 IPs). Assuming the results are always evicted that gives 25 loads and 25 stores or 400 Bytes of I/O, giving an arithmetic intensity \( AI_2 \) of 3.125 FLOPs/Byte or an I/O limit of 75 GFLOP/s, which exceeds the theoretical peak performance of this kernel.

As previously, the IP requires 7 instructions to issue two FP operations giving an arithmetic intensity \( AI_1 \) of \( 2 \div 7 \approx 0.28 \). The theoretical maximum performance with 1.2GHz cores is therefore 0.34 GFLOP/s. We can then give a theoretical peak rate for \( P=64 \) cores, i.e. \( P \times 0.34 \approx 21.9 \) GFLOP/s. With cold caches (cf. Figure 7.10(a)), our experiments show an actual peak of 10.5 GFLOP/s, i.e. 48\% of the maximum for \( P=64 \) cores and \( N=1K \).
Figure 7.10: Performance of the ESF. The working set is $32 \times \text{#problem size}$ bytes.
Figure 7.11: Performance of the matrix-matrix product. The working set is \( \approx 200 \times \text{#problem size} \) bytes.
In fact with a little analysis we see that the performance is limited by the COMA ring bandwidth. For row data, each core will request a cache line every 56 cycles assuming perfect sharing (8 words per row) and there will be no sharing between cores connected to the L2 cache, so the bandwidth at the L2 cache is 4 lines every 56 cycles. For column data, every thread will request a new cache line every 7 cycles as there is no locality of access. Assuming perfect sharing between cores i.e. every line is used for column data by every core before being evicted, then this translates at the L2 cache to one line every 7 cycles or 8 lines every 56 cycles. Total bandwidth demanded by the threads therefore with perfect sharing of row and column data is 12 lines every 56 cycles from the 8 caches connected to a ring or 110 GB/s. As the COMA ring supplies 64 GB/s, the peak performance - on warm caches, cf. Figure 7.11(b) - is limited to 12.2 GFLOP/s. Of course, the performance drops for small problem sizes and large place sizes when there are not enough threads per core to hide all latencies.

```c
thread fft_inner(int le, cpx_t* x) {
    index i;
    int w = i & (le - 1), j = (i - w) * 2 + w, ip = j + le;
    cpx_t t = {
        cos_sin[w].re * x[ip].re - cos_sin[w].im * x[ip].im,
        cos_sin[w].im * x[ip].re + cos_sin[w].re * x[ip].im
    };
    x[ip].re = x[j].re - T.re; x[ip].im = x[j].im - T.im;
    x[j].re = x[j].re + T.re; x[j].im = x[j].im + T.im;
}
thread fft_outer(cpx_t* x, int n, shared int token) {
    index k;
    int le = 1 << (k - 1);
    int t = token;
    create(fid; ; 0; n/2; 1; ) fft_inner(le, x);
    sync(fid);
    token = t;
}
thread fft(cpx_t* x, int n) {
    create(fid; ; 1; log2(n)+1; 1; ) fft_outer(x, n, 0);
    sync(fid);
}
```

**Figure 7.12:** Computation kernel for the 1-D FFT.
Figure 7.13: Performance of the 1-D FFT. The working set is $8 \times \text{#problem size}$ bytes, plus a lookup table.
We conclude this section by looking at the 1-D FFT which code is shown in Figure 7.12. Our μTC implementation performs the forward FFT kernel without the bit reversal phase, as it would be used in a scientific application. The implementation, again, is straightforward. For a problem size N, the outer computation is sequential and the inner computation defines N/2 threads each containing 4 loads, 4 FP \textit{muls}, 3 FP \textit{adds}, 3 FP \textit{subs} and 4 stores. This thread contains 42 instructions, so on one core, the theoretical peak performance assuming latency hiding is 42 cycles per thread, each containing 10 FLOPs. We hence determine the arithmetic intensity $AI_1 = \frac{10}{42} \approx 0.24$. We then derive from it the theoretical maximal performance, i.e. $\text{Perf} = 1.2 \times AI_1 \approx 0.28$ GFLOP/s. The actual observed performance - on one core with cold cache, cf. Figure 7.13(a) - is 0.17 GFLOP/s, or $\approx 60\%$ of the expected maximum. When the number of cores and the problem size increase, the program becomes I/O constrained because of the lack of cache injection. No cache injection means that there are more I/O accesses. With warm caches (cf. Figure 7.13(b)), the observed maximal performance - for $P=64$ and $N=16k$ - is 0.40 GFLOP/s.
7.3 Discussion and conclusion

As already stated in the conclusion of Chapter 6, the major technical contribution of this thesis is the existence of a working SVP compiler. This working compiler enables the compilation of non-trivial and more complex scientific benchmarks; the SVP GCC-4.1 compiler is stable with -O1 optimization flag (major compilation optimizations: dead-code elimination, common subexpression elimination, copy propagation, combining operation, etc.). However, performance is not as good as highly-optimized hand-coded assembly as to be expected. The reason for this is that other optimization flags such -O2 and -O3 are not stable with respect to SVP assumptions. For future work, these optimizations such as instruction rescheduling require further investigation before safe use. Optimizing FP operations per instructions or per bytes of data of I/O would improve performance by reducing the limit dictated by arithmetic intensity computations. Moreover, despite the scientific problems used in this chapter, the specificity of the work undertaken did not allow further investigation with larger problems which will also be put for future work. Further investigation into compiling larger problems would reveal more relevant concurrency issues with compilation, but a lack of time and resources did not permit us to implement larger µTC programs. The Apple-CORE project [90] will provide large benchmarks for the SVP computing system. We did not consider production quality but a proof of concept for a native integration of concurrency assumptions into the compiler’s internals.

Acknowledgments

The author would like to thank Raphael Poss for all the constructive criticisms he made during the elaboration of this chapter with his contribution to the SVP system evaluation section. The author would also like to thank Mike Lankamp for providing the simulator of the target platform (Microgrid) used during evaluation and for his expertise on the hardware perspective of the benchmark analysis.