The Case for Directive Programming for Accelerator Autotuner Optimization[*]

Diana Fayad[1,4], Jakub Kurzak[1], Piotr Luszczek[1], Panruo Wu[1], and Jack Dongarra[1,2,3]

[1] Univerisity of Tennessee
[2] Oak Ridge National Laboratory
[3] Manchester University
[4] University of Paris Saclay

Abstract. In this work, we present the use of compiler pragma directives for parallelizing autotuning of specialized compute kernels for hardware accelerators. A set of constructs, that include prallelizing a source code that prune a generated search space with a large number of constraints for an autotunning infrastructure. For a better performance we studied optimization aimed at minimization of the run time. We also studied the behavior of the parallel load balance and the speedup on four different machines: x86, Xeon Phi, ARMv8, and POWER8.

I. INTRODUCTION

Autotuning has been established as a way of achieving performance for a specific situations often of interest in HPC. Various projects and frameworks provide *autotuners* that allow users to target optimizations for specific computational kernels. The optimal code is, as the name suggests, auto-generated and is not limited to portable constructs but may also include non-portable syntax extensions. The optimization is done through a serious of performance experiments often in model-less fashion but with a use of efficient search heuristics.

BONSAI (Benchtesting OpeN Software Autotuning Infrastructure) is a software project that optimizes the execution of computational kernels on different hardware platforms. The kernels are generated from a user-defined template that is specified in terms of a variety of parameters. The optimal implementation of the kernel is an instantiation of that template and chosen based on fastest execution on the target hardware. The parameter space to be searched for an optimal configuration is defined using the LANAI (LANguage for Autotuning Infrastructure) language.

There are five major steps that lead to the optimal implementation of a kernel:

1. The user defines a template for the kernel that is parametrized with a set of parameters.
2. The user defines the search space of possible parameter combinations and BONSAI generates the code for exploring that space.
3. The code for search space exploration is executed and generates a list of kernels eligible for execution on the accelerator.

[*] This research was supported by NSF through grant 1642441.

4. The eligible kernels generated out of the template and benchmarked for performance on the GPU accelerator.
5. Optionally, the performance data is analyzed and visualized based on the metrics gathered during execution. These metrics include performance, energy, and/or hardware counters. These results could inform the redesign of the original template to improve the performance even further.

This article focuses on #3 that often is time consuming because it deals with the input parameter space, which grows combinatorially with the number of parameters as described in Section III. The problem we solve is described in Section V.

II. Motivation

The main goal of LANAI is to allow the user to specify an exhaustive search space for autotuning parameters, in order to find the kernel with the best performance in terms of time (other metrics such as energy are also possible [1]). Instead of applying arbitrary constraints, LANAI allows the user to base the constraints on the hardware and algorithmic considerations and thus not worry about excluding kernels with optimal performance. The experiment results that we present here explain the importance of efficient exploration of the search space and application of the constraints. In addition to using parallelism, load balancing the threads' workload affects their performance in practical cases of autotuning. We analyze the results based on comparison of multiple directive-based parallelization methods and give a detailed account on how the threads' relative performance.

III. Formal Search Space Description Through Parameter Ranges and Constraints

The search space for autotuning optimization is multidimensional and heterogeneous[5]. It comprises different categories of parameters, namely:

– $\mathbb{B}_i$ binary type, for example: use either vector or scalar operations;
– $\mathbb{O}_j$ ordinal/categorical/enumeration type, for example: use shared memory, texture cache or Level 1 cache;
– $\mathbb{I}_k$ integer type, for example: a range of matrix blocking factors; and
– $\mathbb{C}_\ell$ continuous (represented as floating-point) type, for example: GPU occupancy above 30%.

Theoretically, the entire search space $\mathcal{S}$ may be represented by a Cartesian product of these dimensions:

$$\mathcal{S} = \prod_i \mathbb{B}_i \times \prod_j \mathbb{O}_j \times \prod_k \mathbb{I}_k \times \prod_\ell \mathbb{C}_\ell \tag{1}$$

[5] In order to keep the discussion focused, we will mostly limit the examples to GPU accelerators and their parameters but the concepts are easily translatable to other HPC platforms such as x86 multicore CPUs, Intel Xeon Phi KNL, IBM POWER8 SMT.

	Semantics	LANAI syntax	Generated C code
Iterator	$\mathbb{I}_1 = \{1,3,5,7\}$	`I1 = range(1,8,2)`	`for (int I1 = 1;I1 < 8;I1 += 1)`
Constraint	$\mathbb{E}_1 = \{i_1 \geq 3\}$	`E1 = I1 >= 3`	`if(! I1 >= 3) break;`

Fig. 1. Sample syntax and semantics of LANAI code and the generated C code.

In practice, the search space is much less regular due to a number of software and hardware constraints. the LANguage for Autotuning Infrastructure [1],

In this work, we use LANAI to specify a search space for a typical autotuning task with parameters and constraints derived from a matrix multiplication on a GPU [2]. The specification is written as a Python code. The space is a product of ranges of matrix sizes, block dimensions, threads-per-block counts, and so on. The conditions include maximum sizes per thread-block and the total number of threads etc. LANAI compiler generates a C code, that is used to prune the search space to limit the number of kernel candidates that have to be run on the GPU to assess their performance. The eligible kernel configurations are exported as CSV (Comma-Separated Values) file that is used by the benchtesting module, which is responsible for building and executing the kernels.

For completeness, we show in Figure 1 a quick overview of the syntax, semantics, and the generated code for a simple iterator and a constraint. Full details of the LANAI can be found elsewhere [1].

IV. Related Work

Currently, there is a significant interest in batch matrix operations. Both NVIDIA cuBLAS [3] and Intel MKL [4] include extensive sets of batch routines for basic linear algebra tasks, and so does the MAGMA library from University of Tennessee [5]. Numerous papers have been published about the development and optimization of batch routines [6,7,8,9,10]. This paper is a follow up to our previous work on the batch Cholesky factorization for small matrices [11]. The direct motivation for this work came from the *Alternating Least Squares* (ALS) algorithm for recommender systems [12]. In our previous article, we looked into the development of batch routines for the canonical column-wise data layout. Here we are investigating alternative layouts for batches of extremely small matrices.

Our autotuning methodology is based on the autotuning approach that we pioneered with ATLAS [13] and that now grew into a vibrant field of experimental performance optimization guided by execution profiles. To name a few efforts, we could start with Portable High Performance ANSI C (PHiPAC) [14] that generated code for superscalar processors implementing dense linear algebra operations. Sparse matrix computations were targetted by Optimized Sparse Kernel Interface (OSKI) [15] and FFT and similar transforms were optimized by the Fastest Fourier Transform in the West (FFTW) [16] and Spiral [17]. In fact, Spiral's automated code synthesis has recently addressed matrix-matrix multiply [18]. To our best knowledge, these projects do not target their autotuning efforts specifically for accelerators, and they mostly use the human expert knowledge of the tuning as an embeddable component – the implementation code rather than exposing this knowledge as a generic templates/stencils in the way we do. Also, there are DSLs designed specifically for the purpose of autotuning parallel scientific codes [19,20]. A more exhaustive survey of recent recent advances in the area of autotuning is available elsewhere [21]. The recent work on GPU-specific autotuning [11,1,22] is also relevant.

V. NESTED LOOP PROBLEM

The components of LANAI-based specification define hardware, software, and/or user defined constraints that allow the LANAI compiler to generate the corresponding possible and acceptable configuration for a certain kernel. Figure 2 shows such a generated code from a Python-like code that was used for a matrix matrix multiplication kernel [1]. As we can see in the Figure, the generated code consists of properly nested `for`-loops and several `if`-statements that check conditions embedded inside the loop nest as deep as is allowed by the dependence between the iterators and constraints. In order to understand how the code works, we provide a brief description.

The `for`-loops generate all the possible configurations that a kernel might have while each condition checks their validity and filters the invalid configurations. Thus, a configuration is either **acceptable** and then the iteration continues toward the next iteration of the loop or **unacceptable** and the iteration produces no configuration and `continue` statement moves on to the next iteration. The filtering is based on the user-defined constraints (either software, hardware, or kernel-specific). Due the deep loop nest and non-trivial range of each loop, we could easily predict that such a code would take a long time to execute: typically it has to go through millions of configurations but it commonly filters them down to a few thousands that are used in execution on the accelerator. Our main goal then is to try to parallelize such code and optimize the execution time of LANAI. The natural choice for parallelization is to use pragma directives due their simplicity and minimal effort in inserting them either manually or automatically by the code generator. In order to achieve the best performance, we studied various strategies using a number OpenMP directives and clauses. In particular, we used the following:

- Implementation based on the `critical` construct.
- Implementation based on letting each thread write its own export file.
- Implementation based on writing in a single array allocated for all threads.
- Implementation based on writing in a dynamically allocated/reallocated array of configurations for each thread that is subsequently saved to a file.

Moreover, for each of these strategies, we considered different OpenMP scheduling flavors. We used either static or dynamic scheduling in the parallel-`for` loop construct.

VI. IMPLEMENTATION BASED ON USING CRITICAL CONSTRUCT

VI.1 Static Scheduling Construct

The first intuitive use of parallel-`for` is to use a static scheduling construct that splits the loop iterations over the number of working threads. The attractive behavior of default

```
for (dim_m = 1; dim_m < 1025; dim_m += 1)
 for (trans_a = 0; trans_a < 1; trans_a += 1)
  for (trans_b = 0; trans_b < 1; trans_b += 1)
   for (blk_k = 1; blk_k < (min(1024, 1024) + 1); blk_k += 1) {
    if (dim_vec_delta != 0)
     for (dim_vec = dim_vec_start; dim_vec < dim_vec_bound; dim_vec += dim_vec_delta)
      for (dim_n = 1; dim_n < 1025; dim_n += 1)
       for (tex_b = 0; tex_b < 2; tex_b += 1)
        for (tex_a = 0; tex_a < 2; tex_a += 1) {
         threads_per_block = dim_m * dim_n;
         if (threads_per_block > max_threads_per_block) continue;
         if ((threads_per_block % warp_size) != 0) continue;
         if (vec_mul_delta != 0)
          for (vec_mul = vec_mul_start; vec_mul < vec_mul_bound; vec_mul += vec_mul_delta) {
           if (blk_m_delta != 0)
            for (blk_m = blk_m_start; blk_m < blk_m_bound; blk_m += blk_m_delta) {
             if (blk_n_delta != 0)
              for (blk_n = blk_n_start; blk_n < blk_n_bound; blk_n += blk_n_delta) {
               if (shmem_per_block > max_shared_mem_per_block) continue;
               if (regs_per_thread > max_registers_per_thread) continue;
               if (regs_per_block > max_regs_per_block) continue;
               if ((fmas_per_block / loads_per_block) < min_fmas_per_load) continue;
               if (max_threads_by_shmem < min_threads_per_multi_processor) continue;
               if (max_threads_by_regs < min_threads_per_multi_processor) continue;
               if (dim_n_b_delta != 0)
               for (dim_n_b = dim_n_b_start; dim_n_b < dim_n_b_bound; dim_n_b += dim_n_b_delta) {
                 if (dim_n_a_delta != 0)
                for (dim_n_a =dim_n_a_start; dim_n_a < dim_n_a_bound; dim_n_a +=dim_n_a_delta) {
                   if (dim_m_a_delta != 0)
               for (dim_m_a=dim_m_a_start; dim_m_a <dim_m_a_bound; dim_m_a+=dim_m_a_delta){
                     if (dim_m_b_delta != 0)
               for (dim_m_b=dim_m_b_start; dim_m_b<dim_m_b_bound; dim_m_b+=dim_m_b_delta){
                       if ((dim_m_a * dim_n_a) != threads_per_block) continue;
                       if ((dim_m_b * dim_n_b) != threads_per_block) continue;
                       if (((((blk_k % (dim_m_b * dim_vec)) != 0) || ((blk_n % dim_n_b) != 0))))
                        continue;
                       if (((((blk_m % (dim_m_a * dim_vec)) != 0) || ((blk_k % dim_n_a) != 0))))
                        continue;
                       idx += 1;
                      record(trans_a, trans_b, dim_m, dim_n, blk_m, blk_n, blk_k, dim_vec, vec_mul,
                           dim_m_a, dim_n_a, dim_m_b, dim_n_b, tex_a, tex_b, idx);
                      }}}}}}}}}
```

Fig. 2. Generated C code

static scheduling is that it is straightforward for the OpenMP scheduler since it does not have to maintain a queue of tasks for each thread (in case chunking is enabled). This may sometimes involve complex mechanisms like deadlock handling mechanism, or runtime dynamic distribution of load (in case of using dynamic scheduler). Thus, it is well known that such scheduling is very good when the per-thread workload is equally distributed but it might suffer from load imbalance when different threads have different amount of work.

We did analyze the LANAI code, and we predicted that such a static scheduling construct might not be a good option since we know that some boundary of the loop iteration space might be filtered at an earlier stage of the code and thus its corresponding thread will finish earlier.

We conducted research on where to put the parallel for construct (top, or lower level of the code) with generic heuristic of using outer loops as targets for parallelization. If we put the parallelization at a lower (inner) loop level, there is a big overhead related to the creation of the parallel-`for` instance, and the barrier at the end of each parallel-`for` required for synchronization of the output in the I/O buffers. Nonetheless, we did try such implementation and the results were disappointing and are not reported here. The best implementation is to parallelize the most outer loop which is the first top loop. Using the `collapse` construct, we can see this in Figure 2 as the first four loops are fused together and collapsed because there is no `if`-statement in between. This will create more dynamic behavior at runtime and can contribute to spreading out the workload across the threads.

To give the reader more details about relative workload handled by the individual threads, we performed an experiment with 8 threads. We collected the number of iterations that ended up being assigned to each thread and the number of configurations that each thread generated. In other words, how many iterations of the inner-most loop each thread executed that resulted in generating an acceptable configuration. We illustrated these findings in Figure 3. We also collected the information on the time spent in each thread and show it in Figure 4.

As we could notice from workload and time split figures, Thread 0 ended up with all the work since it spent nearly 109 seconds to finish. Furthermore, it is clear from Figure 3 that the first thread generates all the acceptable configurations while all the other threads terminated their part of the loop-nest (at an earlier stage of the code). As a consequence, it is expected that Thread 0 will be the slowest and the performance will be bound by its speed.

On the other hand, Figure 3 also helps us in understanding the LANAI code behavior. We may conclude that the first portion of the outer loop (the portion that got assigned to Thread 0) is the most important because it generates majority (if not all) of the acceptable configurations. In order to avoid such imbalance, we decided to use the chunk option combined with the static scheduling.

We could use a chunk of size 1 but as we know such a chunk size would create a lot of overhead for the OpenMP runtime scheduler. On the other hand, using too large of a chunk size (as is often the default) might end up effectively serializing the code execution which clearly is not what we want. In the end, it is better to use a chunk size that is only slightly larger than 1 with avoids scheduling all the work into one thread without much runtime overhead. We also collapsed the four outer-most loops (using the `collapse` construct) to have a better distribution of the work since the work will be distributed using the collapsed loops instead of just using the index set of the outer-most loop.

Because of the aforementioned issues, why the parallelism alone is not the only main focus but equally important is the good distribution of work among threads and using all threads effectively is crucial to get a better performance for this autotuner code. In particular, we could see that the Thread 0 was assigned the majority of work-inducing iterations and, consequently, the most running time. That lead us to always follow the threads' workload distribution and address load balancing problems in our parallelization heuristics with pragma directives.

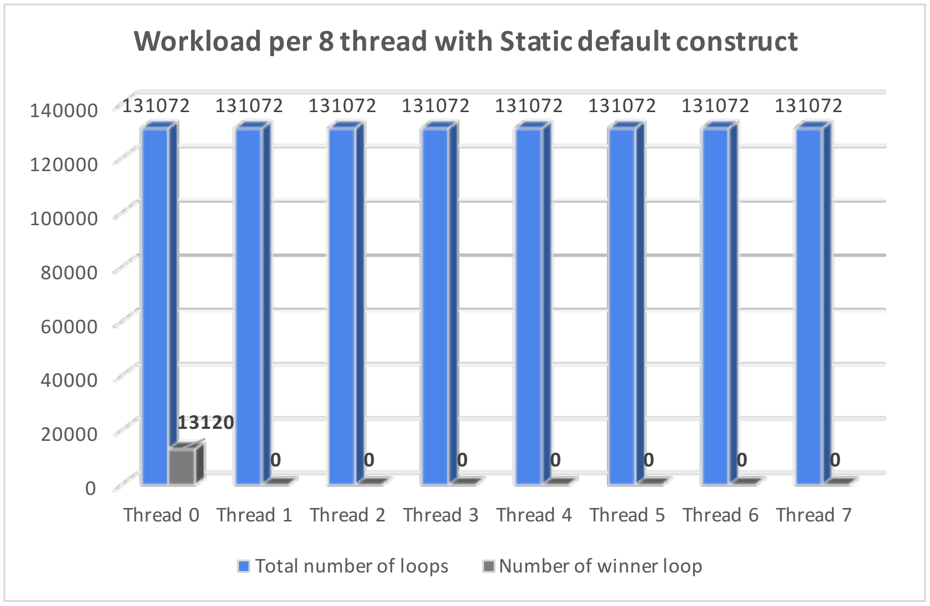

Fig. 3. Workload for 8 threads with static default construct

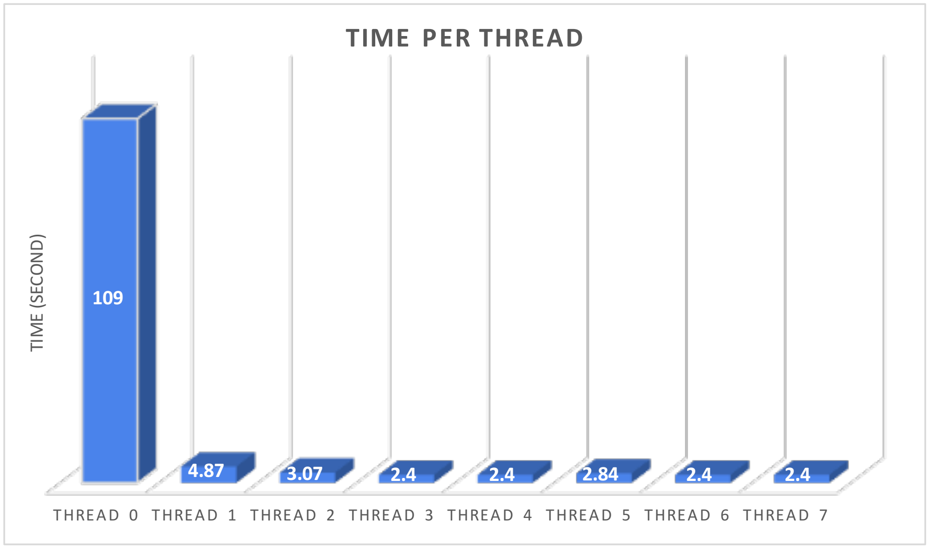

Fig. 4. Time load for 8 threads with static default construct

VI.2 Dynamic Scheduling Construct

As we have mentioned above, using static scheduler is very sensitive to the per-thread workload which relates to the kernel-specific parametrization and constraints – a hard to predict feature of the generated code. As a result, we decided to use dynamic scheduling construct. The goal here is to avoid the predefined (likely static) distribution of work as we are unlikely to know in advance how many iterations each thread will execute because the inner-most loops may be eliminated by the conditions that related to the kernel constraints. The dynamic scheduling construct distributes the load among all threads at runtime and only when each thread requests a work item once it finished the previous work item.

To verify our choice of collapsed loop count in the dynamic scheduling case as well, we performed experiments with and without the `collapse(4)` construct implementation. The use of `collapse(4)` provided better performance across all of our tests. Accordingly, we used this implementation choice for all of the following experiments. This implementation is considered to be very efficient as it asymptotically achieves a perfect speedup as the results below indicate.

VII. Details on the Implementations Used in Tests

Below, we present a number of implementation variants that were appropriate for the `pragma` parallelization.

VII.1 Implementation Based On Letting Each Thread Write Its Own Export File.

In this case, we let each thread write its own export file independently, then we gather them all into one file. Once done, we remove the individual files for each thread. This allows us to eliminate the use of `critical` construct. Note that the gathering happens in the code and it is included in the timing. We performed experiment using the *Multiwriting* strategy. This method with `dynamic` and `collapse` constructs is also a good choice to improve LANAI-generated code as it has similar behavior to the one with `critical` construct.

VII.2 Implementation Based On Writting In A Dynamically Reallocated Table

To avoid using `pragma critical` construct or to write into separate files, we decided to store the data in an array of structures. Each thread defined its own array of structures. The structure contained, beside the array of kernel configuration data, a thread ID, the size of the allocated array and the size actually used. Each thread would reallocate its array of data in case it was needed (in case the size used reached the allocated size).

And after filling all the data in the dynamic arrays, the master thread accessed the individual threads' sructures and subsequently wrote all relevant data in one export file without the need of using critical section or gathering exported files.

VII.3 Implementation Based On Wrtting In A Large Table Allocated Once For All

We could avoid the extra cost of checking the memory space as described in the strategy above with array reallocation, by allocating an array of larger size which ensures that all threads will have no need to reallocate to increase their array sizes. In such a scenario, we could avoid to checking the array size limit. However, such implementation is very sensitive to the characteristic of the kernel in question and also requires either a dry run in advance to get the maximum acceptable configuration. Optionally, a user input might be required that defines the maximum acceptable configuration.

VIII. Performance Results

We first describe the hardware platforms used in the tests.

VIII.1 Knights Landing

Knights Landing (KNL) is the codename for the second-generation of Intel's Xeon Phi many integrated core (MIC) hardware platform. The 7230 and 7250 KNL variants utilize 14 nm lithography similar to the "Broadwell" Xeon E5 and E7 server processors to achieve 3.05 TFLOPS peak double precision floating-point performance. The 68 cores on the 7250 KNL chip are also based upon the 3-way out-of-order "Silvermont" microarchitecture and support four execution threads. Both the 7230 and 7250 KNL variants have two AVX512 vector processing units per core. There are two types of memory used in KNL: a larger DDR4-2400 memory (115.2 GB/s) and a small 16 GB MCDRAM providing up to 490 GB/s of sustained bandwidth through the 2D mesh interconnect. The memory bandwidth per KNL core is approximately 11 GB/sec for small thread counts. In the experiments we used the 16 GB high speed MCDRAM as a NUMA node (flat mode) instead of transparent cache for DRAM (cache mode). We use Quadrant cluster mode. We compiled our codes using GCC 6.1.0.

VIII.2 POWER8

The IBM POWER8 [23] is a RISC microprocessor from IBM fabricated using 22nm technology. The IBM POWER8 processor features up to 12 cores per socket, 8 threads per core, 32 KB instruction cache, 64 KB L1 data cache, 512 KB L2 cache, and 8 MB L3 cache, per core. Each core can issue up to 10 instructions per cycle, and complete 8 instructions per cycle. Functional unit wise, the POWER8 has two independent fixed-point units (FXU), two independent load-store units (LSU) plus two more load units, and two independent floating-point vector/scalar units (VSU). The maximum double-precision floating-point issue rate is 4 fmadds per cycle, single-precision 8 fmadds per cycle. The SIMD width is 2 for double precision and 4 for single precision. Thus the peak double precision performance is Freq $\times$ 8 $\times$ #cores. For a 3.5 GHz frequency 20-cores node the peak DP performance is 560 GFLOPS. We used dual socket IBM POWER8 compute nodes with 20 cores in total. The frequency is about 3.5 GHz. We compiled our codes using GCC 6.3.1.

VIII.3 ARMv8

For this experiment we used a dual socket 96 core Cavium ThunderX compute node. ThunderX features up to 48 cores per socket and is fabricated using 28nm process technology. The clock rate is 2.0 GHz. The double precision fused multiply-add performance is 4 GFLOPS per core. We compiled our codes using GCC 6.1.0. For all the measurements, we used OpenMP environment variables to bind each thread to each hardware core.

VIII.4 Haswell

The Haswell system we uses was an Intel Xeon Processor E5-2650V3 fabricated using 22nm technology. The Haswell processor has 2 sockets with 10 cores per socket and 2 hardware threads per core. It has 32KB for Level 1 data cache, 32K for Level 1 instruction cache, 256K for L2 cache and 25600K Level 3 cache. To compile our codes we used GCC 6.3.0 compiler.

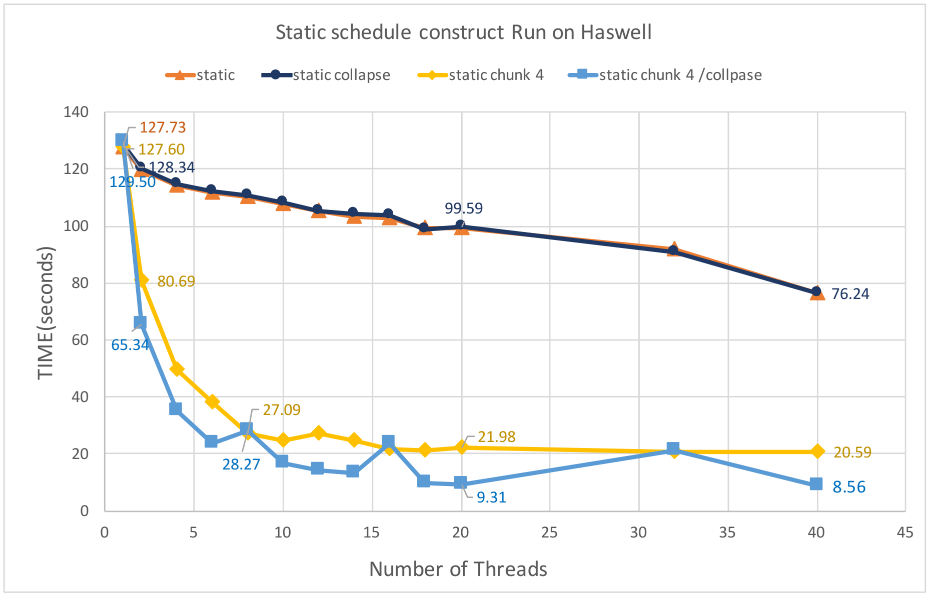

Fig. 5. Static with chunk 4 ***VS*** static default, using collapse 4

VIII.5 Comparison of Static Scheduling Runs

To confirm the analysis we presented above, we did run this version using the `static` construct and plotted the elapsed time in Figure 5. As the figure shows, the time does not decrease linearly when increasing the number of threads The parallel version that uses the `static default` construct or the one that is combined with `collapse`, does not seem to be efficient (for example, using 20 threads takes about 1.3 times faster to finish than the sequential version that requires 128 seconds). Such a low speedup is not acceptable. We found that chunk equal to 4 is a good option. As we can see in the Figure 5, the static scheduler with a chunk equal to 4 is much better than the default one, and this corresponds to the good distribution of work among the threads and avoids excessive load on Thread 0 (this is not the case for the default distribution of iterations).

VIII.6 Dynamic Scheduling Construct

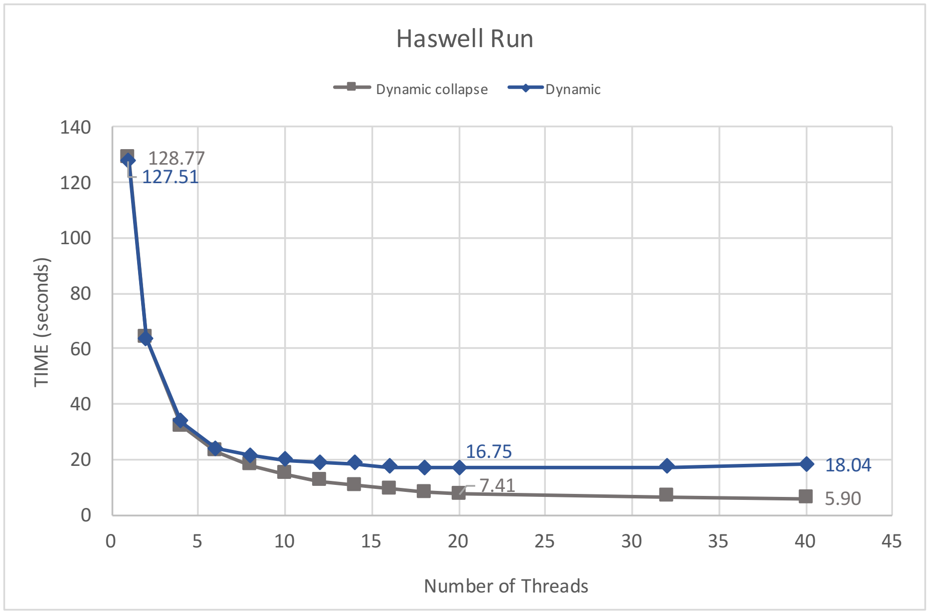

Fig. 6. Dynamic ***VS*** Dynamic collapse 4 run on Haswell

From all the chart drawn in Figure 6, Figure 7, Figure 8, and Figure 9 we can see that the use of dynamic scheduling construct combined with the `collapse` clause provide a very good speedup and distribution of work. For example the on the Haswell machine (Figure 6), using 8 and 20 threads is about 7 and 17 times faster, respectively, than the sequential execution.

VIII.7 Implementation Based on Per-Thread Export File

We can observe in Figure 10 that Multiwriting provides behavior that is similar to the critical strategy with dynamic scheduling. In addition, note the instability of the static schedule for the Threads 8, 16, 32 that corresponds to the pre-distribution of the total number of iterations on the threads.

VIII.8 Implementation Based On Dynamically Allocated Array

As we see in Figure 11, the timing is not much better than the critical and the Multiwriting strategies. Our investigation of the details of this implementation revealed

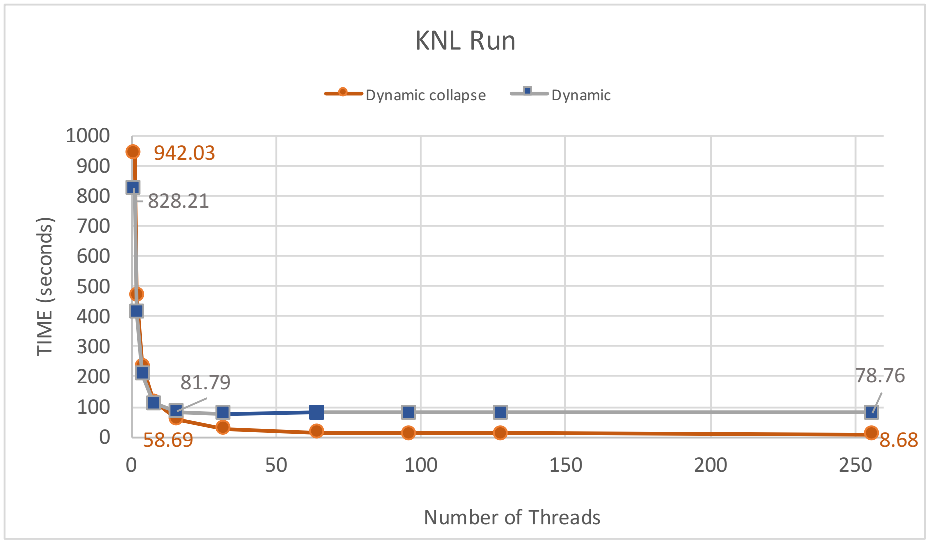

Fig. 7. Dynamic ***VS*** Dynamic collapse 4 run on KNL

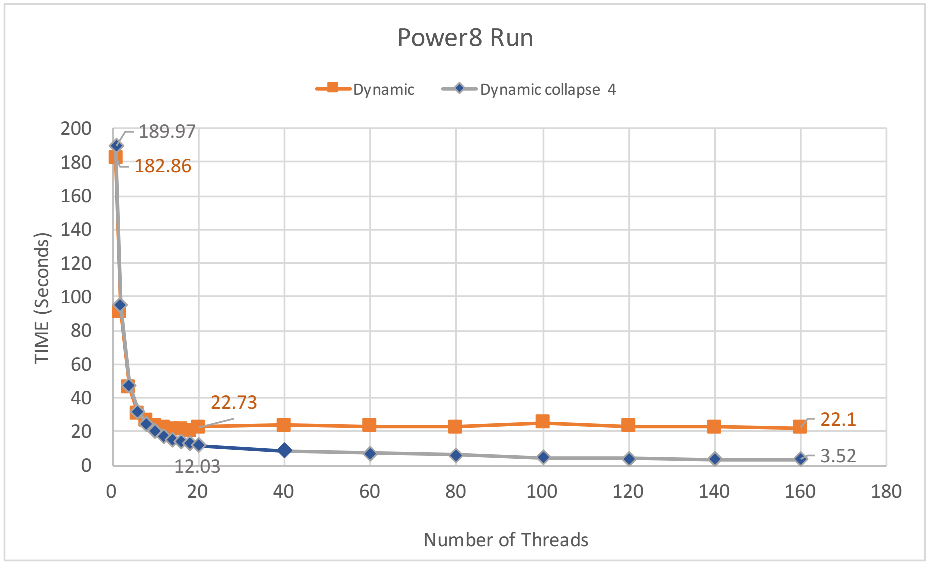

Fig. 8. Dynamic ***VS*** Dynamic collapse 4 run on Power8

that the slowdown of the dynamic reallocation method is due to the necessity of adding an `if`-statements inside the most inner loop, just before filling the array with data. This is done to check whether there is still space available in the allocated array and whether there is a need for a reallocation. Based on this, we decided to avoid the cost of adding

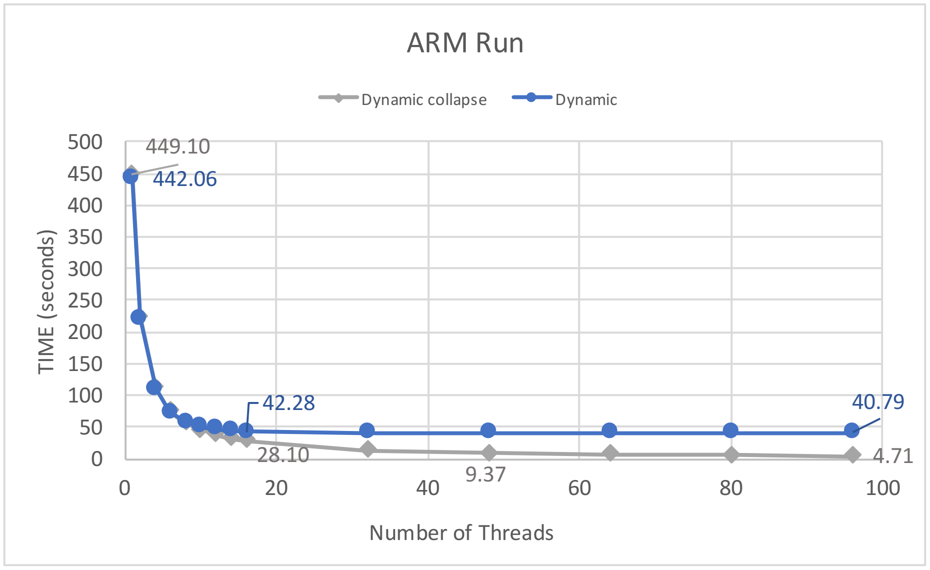

Fig. 9. Dynamic ***VS*** Dynamic collapse 4 run on ARM

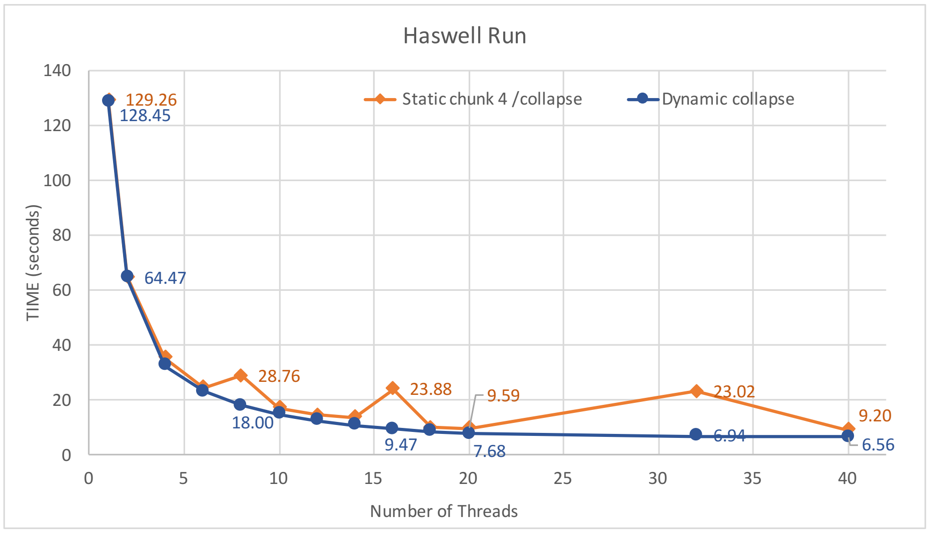

Fig. 10. Multiwriting: Dynamic collapse 4 ***VS*** Static chunk 4 collapse 4 run on Haswell

the code for checking buffer size and instead we allocated a large-enough buffer for each thread's data structure, and we can see the improvement of timing of about 1.5x. We get similar timing for the critical and the Multiwriting strategies.

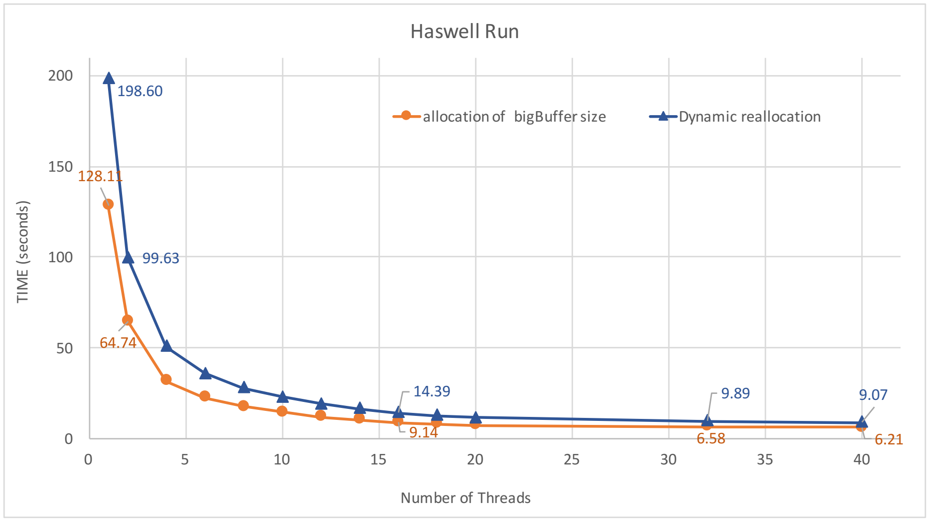

Fig. 11. Reallocate ***VS*** Non-Reallocate array strategy

VIII.9 Analysis of Speedup of the Scheduling Strategies

The speedup of the static scheduling methods implemented in the critical and the Multiwriting strategies were run on the four test machines and results are prsented in Figure 12, Figure 13, Figure 14 and Figure 15. Most strategies appear to be unstable and do not scale linearly. This may be attributed to bad distribution of work for some

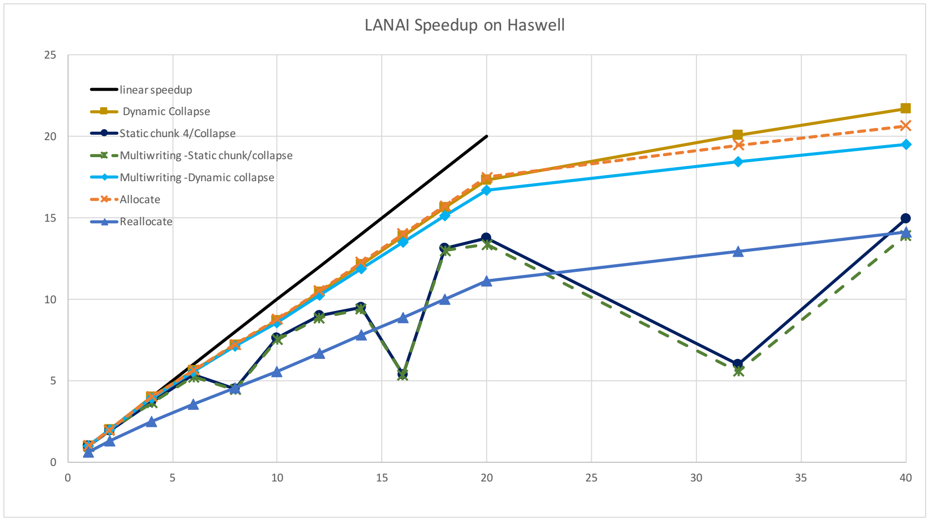

Fig. 12. SpeedUp of LANAI on Haswell

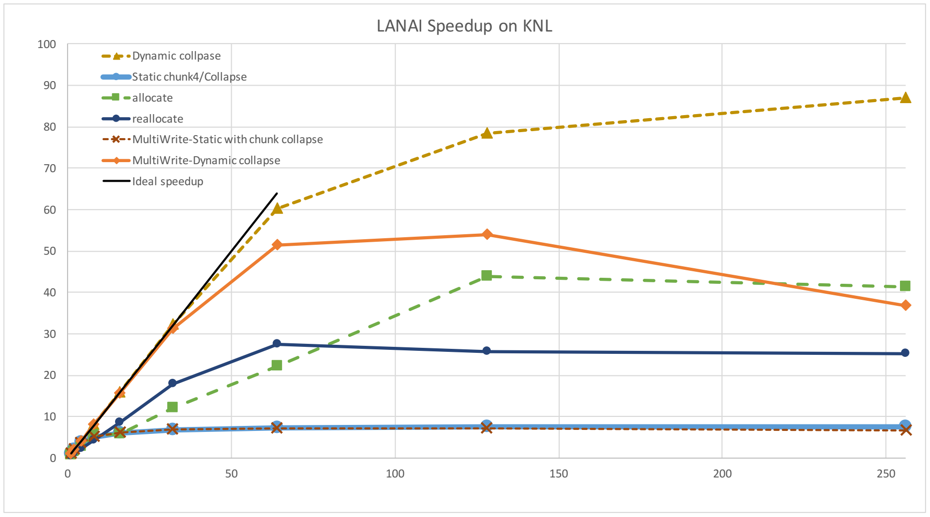

Fig. 13. Speedup of LANAI on KNL

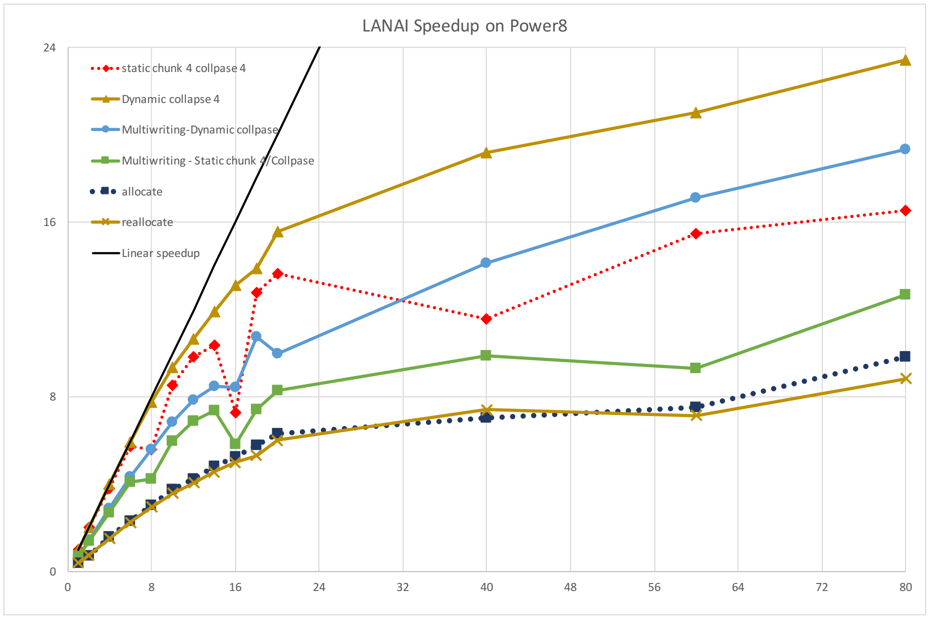

Fig. 14. SpeedUp of LANAI on Power8

threads counts. The known weakness is that the distribution with the static scheduling is done once before the loop starts and threads do not request work as needed and leads directly to a load imbalance for our code. On the other hand, the three methods based on a dynamic scheduling combined with `collapse` (Multiwriting-Allocate-Critical) scale

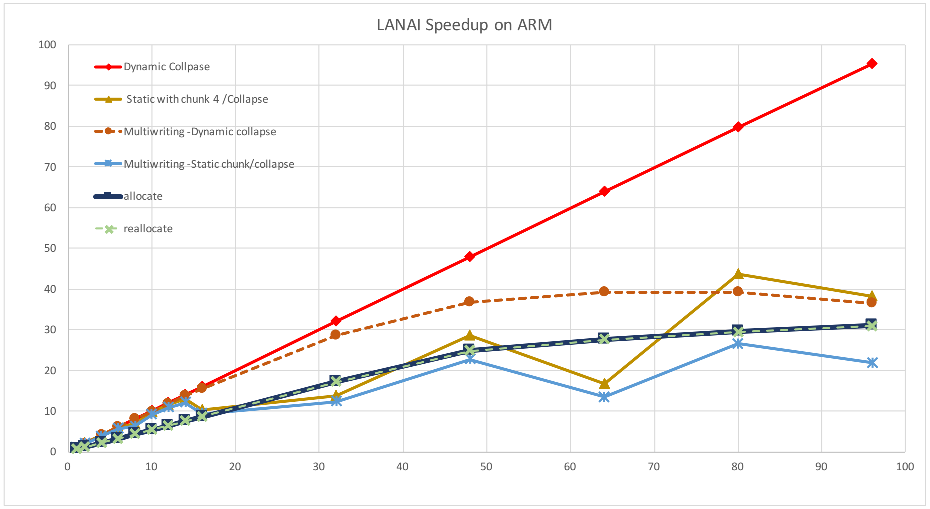

Fig. 15. SpeedUp of LANAI on ARM

linearly. Also, we can see that on the Haswell machine that we see strong scalability until 20 threads and beyond 20 threads we see that the scaling slows down which is mostly due HyperThreading. Similar effect occurs for KNL and POWER8. For KNL, we have for 64 threads a coefficient of 60 and 52 for critical construct and Multiwriting, respectively. And beyond that the scalability slows down. This lack of scaling shows that the hardware cores are not that useful for evaluating search space for the LANAI-generated code. For the ARM machine, there is no hyperthreading. Thus, we get close to linear speedup until 96 threads with the dynamic scheduling method as we see in Figure 15.

Conclusion

We have presented LANAI-generated code that is part of the BONSAI project and used as the autotuning space exploration and pruning. We also showed the importance of using and comparing multiple methods for reducing the run time and watching the threads behavior by analysing the obtained results. After studying the 4 cases of parallel LANAI, that tested various number of threads we concluded that the best result in our experiments in terms of time and threads behavior is the *"Dynamic scheduling using collapse (4) and combined with the critical construct"*. For that setting, we obtained a result of 7.35 seconds for LANAI parallel which is 17 times faster than the LANAI sequential (124 seconds) using 20 threads on the Haswell machine.

References

1. Piotr Luszczek, Mark Gates, Jakub Kurzak, Anthony Danalis, and Jack Dongarra. Search space generation and pruning system for autotuners. In *Proceedings of IPDPSW*, The Eleventh

International Workshop on Automatic Performance Tuning (iWAPT) 2016, Chicago, IL, USA, May 23rd 2016. IEEE.

2. Hartwig Anzt, Blake Haugen, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Experiences in autotuning matrix multiplication for energy minimization on gpus. *Concurrency and Computation, Practice and Experience*, 27(17):5096–5113, December 10 2015.
3. NVIDIA Corporation. *cuBLAS Library User Guide*, DU-06702-001_v8.0 edition, September 2016.
4. Intel Corporation. *Intel Math Kernel Library Developer Reference*, revision: 011 edition, 2017.
5. Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.
6. Oreste Villa, Massimiliano Fatica, Nitin Gawande, and Antonino Tumeo. Power/performance trade-offs of small batched LU based solvers on GPUs. In *European Conference on Parallel Processing*, pages 813–825. Springer, 2013.
7. Tingxing Dong, Azzam Haidar, Piotr Luszczek, James Austin Harris, Stanimire Tomov, and Jack Dongarra. LU factorization of small matrices: Accelerating batched DGETRF on the GPU. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*, pages 157–160. IEEE, 2014.
8. Tingxing Dong, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. A fast batched cholesky factorization on a gpu. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 432–440. IEEE, 2014.
9. Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Towards batched linear solvers on accelerated hardware platforms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 261–262. ACM, 2015.
10. Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Batched matrix computations on hardware accelerators based on gpus. *International Journal of High Performance Computing Applications*, page 1094342014567546, 2015.
11. Jakub Kurzak, Hartwig Anzt, Mark Gates, and Jack Dongarra. Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 27(7):2036–2048, 2016.
12. Mark Gates, Hartwig Anzt, Jakub Kurzak, and Jack Dongarra. Accelerating collaborative filtering using concepts from high performance computing. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 667–676. IEEE, 2015.
13. R. C Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parellel Comput. Syst. Appl.*, 27(1-2):3–35, 2001. http://dx.doi.org/10.1016/S0167-8191(00)00087-9.
14. Jeff Bilmes, Krste Asanović, James W. Demmel, Dominic Lam, and Chee-Whye Chin. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. Technical Report 111, LAPACK Working Note, August 8 1996. University of Tennessee Computer Science Technical Report UT-CS-96-326.
15. Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *SciDAC, J. Physics: Conf. Ser.*, 16:521–530, 2005. DOI: http://dx.doi.org/10.1088/1742-6596/16/1/071.
16. Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. special issue on ”Program Generation, Optimization, and Adaptation”.

17. M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005. http://dx.doi.org/10.1109/JPROC.2004.840306.
18. Richard Veras and Franz Franchetti. Capturing the expert: Generating fast matrix-multiply kernels with spiral. In Michel Daydé, Osni Marques, and Kengo Nakajima, editors, *High Performance Computing for Computational Science – VECPAR 2014, The Ninth International Workshop on Automatic Performance Tuning (iWAPT)*, pages 236–244, 2014.
19. Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. Atune-IL: An instrumentation language for auto-tuning parallel applications. In Springer Berlin / Heidelberg, editor, *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, volume LNCS, pages 9–20, August 2009.
20. Shoaib Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages*. PhD thesis, University of California, Berkeley, 2012. Tech Report EECS-2012-255.
21. S. Benkner, F. Franchetti, M. Gerndt, and J. K. Hollingsworth. Automatic application tuning for hpc architectures. *Dagstuhl Reports*, 3(9):214–244, January 2014. http://dx.doi.org/10.4230/DagRep.3.9.214; http://drops.dagstuhl.de/opus/volltexte/2014/4423.
22. Mark Gates, Jakub Kurzak, Piotr Luszczek, Yu Pei, and Jack Dongarra. Autotuning batch Cholesky factorization in CUDA with interleaved layout of matrices. In *Proceedings of IPDPSW*, The Twelth International Workshop on Automatic Performance Tuning (iWAPT) 2017, Orlando, FL, USA, May 2017. IEEE.
23. Balaram Sinharoy, JA Van Norstrand, Richard J Eickemeyer, Hung Q Le, Jens Leenstra, Dung Q Nguyen, B Konigsburg, K Ward, MD Brown, José E Moreira, et al. Ibm power8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):2–1, 2015.