Massively Parallel Automated Software Tuning

Jakub Kurzak
University of Tennessee
Knoxville, Tennessee
kurzak@icl.utk.edu

Yaohung M. Tsai
University of Tennessee
Knoxville, Tennessee
ytsai2@vols.utk.edu

Mark Gates
University of Tennessee
Knoxville, Tennessee
mgates3@icl.utk.edu

Ahmad Abdelfattah
University of Tennessee
Knoxville, Tennessee
ahmad@icl.utk.edu

Jack Dongarra*
University of Tennessee
Knoxville, Tennessee
dongarra@icl.utk.edu

ABSTRACT

This article presents an implementation of a distributed autotuning engine developed as part of the Bench-testing OpenN Software Autotuning Infrastructure project. The system is geared towards performance optimization of computational kernels for graphics processing units, and allows for the deployment of vast autotuning sweeps to massively parallel machines. The software implements dynamic work scheduling to distributed-memory resources and takes advantage of multithreading for parallel compilation and dispatches kernel launches to multiple accelerators. This paper lays out the main design principles of the system and discusses the basic mechanics of the initial implementation. Preliminary performance results are presented, encountered challenges are discussed, and the future directions are outlined.

CCS CONCEPTS

• **Software and its engineering → Massively parallel systems**.

KEYWORDS

automated software tuning, graphics processing unit

ACM Reference Format:
Jakub Kurzak, Yaohung M. Tsai, Mark Gates, Ahmad Abdelfattah, and Jack Dongarra. 2019. Massively Parallel Automated Software Tuning. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337908

1 INTRODUCTION

The goal of the Bench-testing OpeN Software Autotuning Infrastructure (BONSAI) project is to develop a toolset for facilitating huge autotuning sweeps of computational kernels for graphics processing units (GPUs). This article describes BONSAI's distributed bench-testing engine for deploying large tuning sweeps to massively parallel distributed-memory machines with multiple GPUs per node. Such machines are becoming ubiquitous in scientific and engineering computing and right now seem to be the only viable path toward reaching exascale. At the same time, autotuning is one of the main performance engineering tools for GPU kernel development. Also, as we further discuss in Section 2, there is a clear need for deploying really massive sweeps. Yet, no software infrastructure exists for utilizing the vast available resources in the autotuning process.

One important factor is the relation between autotuning and machine learning. While machine learning is commonly applied to the problem of performance tuning, BONSAI's ability to produce exhaustive sweeps creates a unique opportunity to establish the ground truth for the ultimate validation of different machine learning approaches. Not without significance is the growing popularity of deep learning and its dependence on the availability of large datasets for training. BONSAI offers vast possibilities for applying deep neural networks (DNNs) to the problem of automated software tuning in new creative ways by allowing us to produce input datasets of unprecedented sizes for the training phase.

2 MOTIVATION

We argue that the current trends in high-end computing can be judged by taking a look at the largest systems on the TOP500 list. [1] Consider the architecture of the number one system on the list, the Summit supercomputer at the Oak Ridge Leadership Computing Facility (OLCF). [2] Summit contains three NVIDIA V100 GPUs per each POWER9 CPU. The peak double-precision floating-point performance of the CPU is 22 $(cores) \times 24.56\ GFLOPS = 540.32\ GFLOPS$. The peak performance of the GPUs is 3 $(devices) \times 7.8\ TFLOPS = 23.4\ TFLOPS$. I.e., 97.7% of performance is on the GPU side, and only 2.3% of performance is on the CPU side. This means that maximum offload to GPUs is paramount and aggressive optimizations are crucial. This also means that vast numbers of GPUs are readily available for the process of autotuning. Consider that to place the Summit system on the TOP500 list the High-Performance Linpack (HPL) benchmark[5] had to run for approximately 9 hours using 27,648 GPUs. If the resources were used instead for a tuning sweep, it would be equivalent to over 28 years of serial tuning using one GPU.

*Jack Dongarra also holds appointments at Oak Ridge National Laboratory and the University of Manchester.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP 2019, August 5–8, 2019, Kyoto, Japan
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337908

[1] https://www.top500.org
[2] https://en.wikichip.org/wiki/supercomputers/summit

The question remains whether such massive sweeps are justified. Here we try to make the case for such runs. One of the prime examples is the omnipresent matrix multiplication kernel, also known as the `gemm` routine in linear algebra lingo. Consider an implementation of the `gemm` kernel using NVIDIA's CUDA [9, 12] platform. The kernel can have up to nine blocking factors, can be implemented with or without the use of single instruction, multiple data (SIMD) instructions, can be implemented with or without the use of texture memory, can be invoked with two values for the `cudaFuncCache` setting and two values for `cudaSharedMemConfig` setting.[3] In addition, the kernel is needed in single precision and double precision, and in real arithmetic and complex arithmetic. On top of that, it needs to support four combination of the `transa` and `transb` input arguments.[4] Finally, the performance of `gemm` is sensitive to the sizes of the input matrices, and ideally would be tuned for a range of different sizes, which adds three more dimensions (m, n, k) to the search space.

Another important issue is the potential sensitivity of the kernel being tuned to the input data, the prime example here being the sparse matrix-vector product kernel, also known as the `SpMV` routine. Performance of `SpMV` is sensitive to the contents of the input matrix and the layout used for its representation. Arguably, it is not without merit to run a performance sweep over all the 2,833 matrices in the SuiteSparse collection[5] using a number of different layouts (CSC/CSR, ELLPACK, SELL-C/SELL-P, Sell-C-Sigma, BCSR, DIA, COO, etc.) This alone creates tens of thousands of cases, without even considering tunable parameters of the kernels' implementations.

It is important to note, though, that we are not dismissing the merits of intelligent pruning of the search space. Notably, BONSAI contains a dedicated component for search space generation and pruning [10]. The argument is that pruning can be limited to the cases that are guaranteed to fail—either during compilation, launch, or execution—or are absolutely certain to deliver inferior performance. The point is to eliminate the educated guesswork from the tuning process forced by superficial constraints on the size of the tuning ensemble.

Finally, an interesting avenue is the collection of data from the hardware performance counters. Consider that, for devices with compute capability 7.x, the `nvprof` tool from NVIDIA can collect 62 performance events, which can be used to calculate 176 performance metrics. [6] These metrics provide a plethora of information, such as: achieved occupancy, instructions executed per cycle, shared memory efficiency, etc. Computing all the metrics in the course of a performance tuning sweep may produce a great dataset for machine learning algorithms. At the same time, only a handful of events can be collected in a single kernel launch, and the kernel has to be "replayed" multiple times to collect all of them, which leads to excessive execution times.

[3] https://docs.nvidia.com/cuda/cuda-runtime-api/ → Device Management
[4] https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-gemm
[5] https://sparse.tamu.edu/about
[6] https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference-7x

3 ORIGINAL CONTRIBUTION

We believe that BONSAI's distributed bench-testing system is a unique solution, specifically when considering its dynamic scheduling capabilities when it comes to:

- dynamic dispatch of work to distributed-memory nodes,
- dynamic dispatch of CPU tasks to multiple CPU cores,
- dynamic dispatch of GPU tasks to multiple GPU devices.

It is the only system that we know of that directly targets massively parallel machines for parallel compilation as well as launches of a large number of GPU kernels.

It needs to be pointed out that the system implements parameter sweeps, which are in principle embarrassingly parallel. Notably, in the data analytics space, Java-based frameworks such as Hadoop [7] and Spark [8] are mainstream solutions for such workloads. However, in our opinion, the listed capabilities would not easily be realized using such software. An additional hurdle would be the fact that Hadoop and Spark are not trivial do deploy to supercomputer installations and usually not available at supercomputing sites.

4 RELATED WORK

The body of work on automated software tuning is vast. The pioneering efforts for tuning dense matrix kernels were the Automatically Tuned Linear Algebra Software (ATLAS) [17], and its predecessor, the Portable High Performance ANSI C (PHiPAC) [3]. Seminal work on tuning sparse matrix kernels was done in the Optimized Sparse Kernel Interface (OSKI) project [16]. Autotuning signal processing kernels was spearheaded by the Fastest Fourier Transform in the West (FFTW) package [6] and the Signal Processing, Imaging, Reasoning, and Learning (SPIRAL) project [13]. The PATUS [4] and Sepya [7] projects championed autotuning techniques for stencil computations.

While autotuning was pioneered by domain-specific techniques, more general solutions also emerged to target a wider range of applications. Seminal work in this area was done almost two decades ago in the Active Harmony project [15] and the FIBER project [8]. More recently, the same objective was targeted by the OpenTuner system [1] and the ppOpen-HPC framework [11].

There is a vast body of work on autotuning kernels for GPU accelerators, ranging from dense linear algebra to sparse linear algebra, to signal processing, stencil codes, etc. It is impossible to do justice to all the different activities. Instead, we would like to single out the recent work by David Tanner as a rare example of a conference paper documenting autotuning efforts by the hardware industry [14].

[7] https://hadoop.apache.org
[8] https://spark.apache.org

5 IMPLEMENTATION

BONSAI is implemented as a header-only C++ library and relies on MPI for message passing and OpenMP for multithreading. Currently, it only supports NVIDIA devices using the CUDA programming model. We use CUDA terminology throughout the text and make frequent references to CUDA documentation. We plan to extend the coverage to devices from AMD and Intel in the future.

5.1 Principles of Operation

Figure 1 illustrates BONSAI's principles of operation. The user is responsible for creating three files:

(1) the kernel file, implementing the GPU kernel to be tuned,
(2) the driver file, for setting up the tuning sweep using the BONSAI distributed bench-testing engine, and
(3) the comma-separated values (CSV) input file containing the definition of the search space to traverse.

The BONSAI engine deploys the tuning sweep to a distributed-memory system with GPU accelerators and produces the output CSV file with the results.

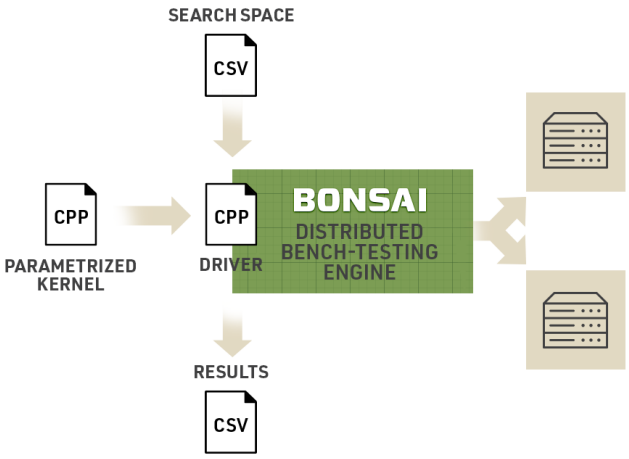

Figure 1: Main principle of BONSAI operation.

The kernel file contains the `__global__` function, implementing the GPU kernel, and the host function, which launches the kernel using the `<<<...>>>` semantics. The kernel can be parametrized using a combination of compile-time parameters and runtime parameters. Compile-time parameters are symbols defined by passing the `-D` flag to the compiler. Consider the canonical matrix multiplication example from the CUDA Programming Guide[9] (the `MatMul` kernel). The code contains the preprocessor directive `#define BLOCK_SIZE 16`. We can remove it from the source code and pass it as a compilation option, `-DBLOCK_SIZE=16`. A different mechanism is used for passing runtime parameters, as explained further later in the text.

Figure 2 shows an example of a CSV input file describing the search space. The first line contains the names of the parameters.

[9]https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory

The second line contains the data types of the parameters. The supported parameters' types are:

Integer - integral values represented internally as `int64_t`,
Real - real values represented internally as `double`,
String - strings represented internally as `char[8]`.

The third line specifies whether the parameter is a compile-time parameter (`Compile`) or a runtime parameter (`Runtime`). The remaining lines contain different sets of values to be tried in the tuning sweep. In the file in Figure 2, `BLOCK_SIZE` is an integer, compile-time parameter, taking values 16, 32, 64, and 128, and `SharedMemConfig` is a string, runtime parameter, taking values `Four` and `Eight`.

```
1  BLOCK_SIZE, SharedMemConfig
2  Integer   , String
3  Compile   , Runtime
4          16, Four
5          16, Eight
6          32, Four
7          32, Eight
8          64, Four
9          64, Eight
10        128, Four
11        128, Eight
```

Figure 2: Example of a CSV input file (indentation and whitespaces added for clarity).

The output CSV file (example shown in Figure 3) is created by adding output parameters to the contents of the input CSV file. By default, the parameters `Status`, `Error`, and `Time` are added. `Status` is a string parameter, indicating a `Success` or a `Failure`. `Error` is a string parameter, indicating the reason for the failure if a failure occured, and can take the following values:

None - no failures occurred,
Compile - compilation failed,
Launch - launch failed,
Test - user-defined test failed.

`Time` is a real-valued parameter representing the execution time in seconds. Figure 3 also contains a real-valued, user-defined parameter `Gflops`. The mechanism for adding user-defined parameters is explained further later in the text.

```
1  BLOCK_SIZE, SharedMemConfig, Status , Error  , Time       , Gflops
2  Integer   , String         , String , String , Real       , Real
3  Compile   , Runtime        , Output , Output , Output     , Output
4          16, Four           , Success, None   , 0.00417039, 491.174
5          16, Eight          , Success, None   , 0.00417222, 490.959
6          32, Four           , Success, None   , 0.00407581, 526.885
7          32, Eight          , Success, None   , 0.00408248, 526.025
8          64, Four           , Failure, Launch , 0          , 0
9          64, Eight          , Failure, Launch , 0          , 0
10        128, Four           , Failure, Compile, 0          , 0
11        128, Eight          , Failure, Compile, 0          , 0
```

Figure 3: Example of a CSV output file (indentation and whitespaces added for clarity).

Figure 3 shows the output from an actual BONSAI execution of the MatMul example from the CUDA Programming Guide. The two cases where `BLOCK_SIZE=64` fail at launch due to exceeding

the maximum number of threads per block, while the two cases where `BLOCK_SIZE=128` fail in compilation due to exceeding the size of the shared memory.

5.2 Processing Pipeline

The purpose of the driver file is to set up the tuning sweep by connecting the other components—the kernel file and the input CSV file—and by establishing the processing pipeline that each record in the input file will go through. Figure 4 shows a driver for tuning the MatMul kernel. First, an object of class `Sweep` is created with a fairly long list of parameters passed to the constructor to describe the setup. The role of each parameter is explained in the comments. Then the `Sweep::addParameter()` method is used to add the extra parameter `Gflops` to the list of output parameters. What follows is a sequence of calls to the `Sweep::addCallback()` method, which creates the pipeline for processing the input records. Finally, the `Sweep::run()` method launches the sweep to the available resources.

```
try {
    using namespace bonsai;
    using Target = Callback::Target;
    using Type   = Callback::Type;

    bonsai::Sweep sweep(
        std::string("input.csv"),     // input CSV filename
        std::string("output.csv"),    // output CSV filename
        std::string("kernel.cu"),     // kernel source filename
        std::string("MatMul"),        // kernel function name
        sizeof(CallbackData),         // size of callback data
        num_reps,                     // number of repetitions
        chunk_size,                   // dispatch chunk size
        std::string("-arch=sm_60")); // extra compilation flags

    sweep.addParameter(std::string("Gflops"), Value::Type::Real);

    sweep.addCallback(init_input,  Target::Host);
    sweep.addCallback(copy_input,  Target::Device);
    sweep.addCallback(call_kernel, Target::Device, Type::Kernel);
    sweep.addCallback(copy_output, Target::Device);
    sweep.addCallback(free_mem,    Target::Device, Type::Cleanup);
    sweep.addCallback(test_result, Target::Host,   Type::Test);

    sweep.run();
}
catch (bonsai::Exception& e) {
    std::cerr << e.what() << std::endl;
}
```

Figure 4: Example of a driver file showing setup and launch of a BONSAI sweep.

Figure 5 shows the execution produced by the driver in Figure 4. First, BONSAI compiles the kernel using the `system()` call to invoke the `nvcc` compiler. The values of the compile-time parameters are passed as preprocessing options, e.g., `-DBLOCK_SIZE=16`. The kernel is compiled to a dynamic library, which is then opened using `dlopen()`, and the appropriate symbol is loaded using `dlsym()`.

Then control is passed to the user-defined callbacks. The callbacks have two attributes: `Target` and `Type`. `Target` can be either `Host` or `Device`.

Host designates a function to be executed by a CPU thread.
Device designates a function to be executed by a GPU.

One important difference between the two classes is that CPU callbacks are executed using OpenMP multithreading (tasking to be exact), with disregard for resource contention, while GPU callbacks are granted exclusive access to one of the GPUs in the system for the duration of their execution, in order to avoid any potential interference.

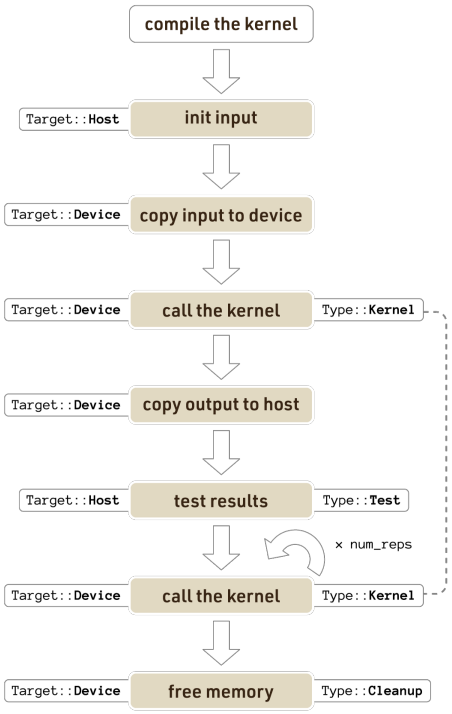

Figure 5: Example of a BONSAI kernel processing pipeline.

The optional `Type` attribute can take the following values:

Kernel designates the callback invoking the GPU kernel.
Test designates the user-supplied correctness check.
Cleanup designates the cleanup callback.

In the initial pass, each callback is called once, with the exception of `Cleanup`, which is skipped. `Kernel` is invoked once, but its execution time is ignored. If present, `Test` is called and its result (`bool`) is noted. If `Test` returns `false`, then, in the output file, `Status` is set to `Failure` and `Error` is set to `Test`. When the initial pass is finished, `Kernel` is invoked `num_reps` times and the best time is taken. Only then `Cleanup` is called—at the very end.

Figures 6 and 7 show excerpts of the callback functions for the MatMul example. Currently, for simplicity, all callbacks have the same signatures and take the following parameters:

Callback data is a block of memory for sharing data among callbacks. It is allocated by BONSAI according to the size specified in the `Sweep()` constructor, and is private to one pass of the pipeline, i.e., one set of input parameters—one line from the input CSV file. Typical usage of the callback data is for passing pointers to input/output arrays that are allocated in one callback, used in another callback, and freed in the cleanup callback.

Kernel parameters is a map providing access to the values of the parameters in a given record. The key is a string with the name of the parameter, e.g., `BLOCK_SIZE`, `SharedMemConfig`. The value is a union of:

```
double real;
int64_t integer;
char string[8];
```

Kernel function is the function invoking the GPU kernel.

CUDA stream is the stream, created by BONSAI, that is dedicated to this particular pass. All CUDA calls in all `Device` callbacks have to be issued to that stream to guarantee exclusive access to one particular GPU.

Figure 6 shows an excerpt of the kernel callback. The value of the `SharedMemConfig` parameter is accessed and passed to the function invoking the kernel, along with the CUDA stream. That function will call `cudaFuncSetSharedMemConfig()` to set the configuration and launch the kernel using the <<<...>>> notation.

```
// Launches the GPU kernel (Target::Device, Type::Kernel)
bool call_kernel(void* callback_data,
                 bonsai::KernelParams& kernel_params,
                 bonsai::Kernel::Func kernel_func,
                 cudaStream_t stream)
{
    cudaSharedMemConfig shm_config;
    char* shm_param =
        kernel_params[std::string("SharedMemConfig")].string;

    if (strcmp(shm_param, "Four") == 0)
        shm_config = cudaSharedMemBankSizeFourByte;
    else
        shm_config = cudaSharedMemBankSizeEightByte;
    ...
    cudaError_t status = kernel_func(shm_config, ..., stream);
    return status == cudaSuccess;
}
```

Figure 6: Excerpt of the kernel callback function.

Figure 7 shows an excerpt of the cleanup callback. This callback frees all the previously allocated memory and sets the value of the user-defined `Gflops` parameter. By the time cleanup is called, the kernel's best execution time is available through the `Time` parameter. The GFLOPS value is computed by dividing the number of floating-point operations by the execution time and dividing by 10^9, and it is then stored in the map of parameters. When the sweep completes, it will be used to populate the Gflops column in the output CSV file.

```
// Performs cleanup tasks (Target::Device, Type::Cleanup)
bool free_mem(void* callback_data,
              bonsai::KernelParams& kernel_params,
              bonsai::Kernel::Func kernel_func,
              cudaStream_t stream)
{
    cudaFree(...);
    ...
    free(...);
    ...
    double time = kernel_params.at(std::string("Time")).real;
    double gflops = ... / time / 1000000000.0;
    kernel_params[std::string("Gflops")].real = gflops;
    return cudaGetLastError() == cudaSuccess;
}
```

Figure 7: Excerpt of the cleanup callback function.

5.3 File IO

BONSAI performs file system operations when:

- reading the input CSV file,
- writing the output CSV file,
- compiling the kernel to a dynamic library, for each record in the input CSV file.

The input CSV file is read by one MPI rank and sent to all the other ranks using a sequence of calls to `MPI_Bcast()`. The input is replicated on all ranks, which massively simplifies dynamic scheduling of work to ranks, as further explained in Section 5.4. At the same time, replication of the entire input is not a problem from the standpoint of memory overhead. Take, for example, an input file with 100 parameters and one million records. Parameters' values are represented as a union of `int64_t`, `double`, and `char[8]`. I.e., each parameter occupies 8 bytes. One million records, 800 bytes each, requires ~0.75 GB. Consider that, e.g., the Summit supercomputer has 512 GB of memory per node. Also, a broadcast of less than one GB of data is not a challenge from the standpoint of MPI communication. The same approach in reverse is followed for writing the output CSV file. The outputs from all the nodes are combined, using a sequence of calls to `MPI_Reduce()`, and one rank takes care of writing the output CSV file.

The operations regarding the CSV files only happen at the beginning and at the end of the execution and are not performance critical. On the other hand, large numbers of compilations performed on each node are performance critical and some caution needs to be taken to avoid file system contention for large runs. Ideally, all compilations happen in the local disk of each node. Usually, a "scratch" folder is created in the file system for access to the local disk in each node, and is sometimes purged when the process completes. The user has a few options to provide its location to BONSAI:

(1) The path can be provided as an optional parameter to the `Sweep()` constructor.
(2) If not passed to the constructor, then the value of the environment variable `$TMPDIR` is used.
(3) If not set in `$TMPDIR`, then the folder `/tmp/` is used.

The kernel source file is copied to the scratch folder at the beginning of the sweep and removed at the end. All the dynamic libraries created over the course of the sweep—one for each record in the input file—are stored in the scratch folder with the row numbers attached to the names to avoid name collisions.

5.4 Parallel Dispatch

Processing of records is scheduled to distributed-memory nodes dynamically at runtime, in chunks, basically the same way that

```
#pragma omp parallel for schedule(dynamic, chunk)
```

would schedule a loop to multiple threads in shared memory. Dynamic scheduling is necessary because of wildly fluctuating compilation and execution times. The time it takes to compile the kernel—with a given set of compile-time parameters—may be anywhere between a fraction of a second and tens of seconds. We have even encountered cases when the compilation exceeded a minute. This usually happens when the compiler is bogged down by excessive unrolling.

Similarly, the kernel's execution times are highly unpredictable due to the very nature of autotuning. We are trying a large number of cases, precisely because we do not know which ones will perform well. An order of magnitude performance fluctuations are to be expected.

Chunking serves a dual purpose. It allows for minimizing the overheads of dispatching work to multiple distributed-memory nodes, while at the same time maximizing the benefits of dynamic scheduling within each node. Ideally, the number of records in the chunk should be much larger than the number of CPU cores and GPU devices in each node. At the same time, the overall number of chunks should be much larger than the number of distributed-memory nodes.

The distributed-memory scheduling in BONSAI follows the traditional client-server architecture. Figure 8 shows the basic structure. Rank 0 dedicates one thread to serving requests for work from other ranks, and dedicates all the other threads to local processing of chunks. The `omp master` section at line 7 contains the server code. The `omp single` section at line 12 contains the local processing code. Each record in the chunk is scheduled as an independent OpenMP task using the `omp task` directive. The `omp master` section at line 21 contains the client code that requests chunks from the server and processes them, in a similar fashion, by executing each record as an independent task.

```
1  void Sweep::run()
2  {
3      if (mpi_rank_ == mpi_root_) {
4          #pragma omp parallel
5          {
6              if (mpi_size_ > 1) {
7                  #pragma omp master
8                  {
9                      // server code
10                 }
11             }
12             #pragma omp single
13             {
14                 // local processing
15             }
16         }
17     }
18     else {
19         #pragma omp parallel
20         {
21             #pragma omp master
22             {
23                 // client code
24             }
25         }
26     }
27 }
```

Figure 8: Top level structure of the scheduling code.

Figure 9 illustrates the code executed by rank 0. The right side shows the server code; the left side shows the local processing code. The server follows the cycle of receiving a request, sending a response, and advancing the chunk counter until the records are exhausted. When that happens, the server sends a termination flag to all the other ranks. The local processing code simply processes chunks and advances the chunk counter until the records are exhausted. Atomic access to the chunk counter is protected by the `omp critical` directive.

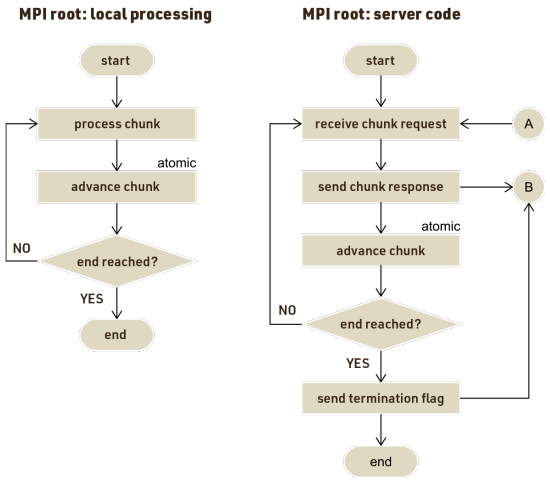

Figure 9: Flowcharts of operation of rank 0.

Figure 10 illustrates the code executed by all ranks, other than 0. They follow the cycle of sending a request, receiving the response, and processing the chunk indicated in the response until the termination flag is received.

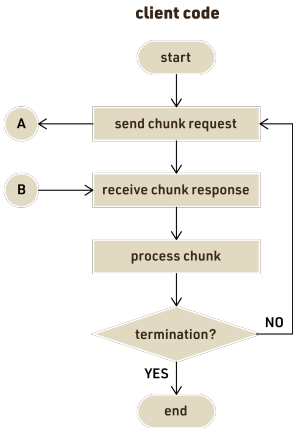

Figure 10: Flowcharts of operation for ranks other than 0.

A chunk is requested by sending an empty message to the server. The server receives the message using `MPI_ANY_SOURCE` and finds out the requester from the `MPI_SOURCE` field in the message's status. It then responds by sending the index of the first record in the next available chunk.

The last aspect of scheduling is the necessity to guarantee exclusive access to devices. While one kernel is executed, timed, and possibly profiled, the GPU should not be executing any other operations—other kernels, memory copies, etc.—which would introduce performance interference. In the course of processing a record, a device is requested after compilation and released after cleanup. Each record has access to one GPU at a time. This mechanism is implemented by using a boolean flag and accessing it

using atomic operations `__sync_bool_compare_and_swap()` and `__sync_lock_test_and_set()`. Processing is blocked until a free device is found. Also, as has already been mentioned, each record executes using a unique CUDA stream.

5.5 Code Structure

BONSAI's distributed engine is implemented by a few C++ classes. The current implementation is header-only. Figure 11 shows the relationships among the main classes. The code also includes a handful of classes for representing simple structures, such as parameters and their values, as well as classes for handling auxiliary tasks, such as exception handling and tracing (printing Gantt charts of the execution).

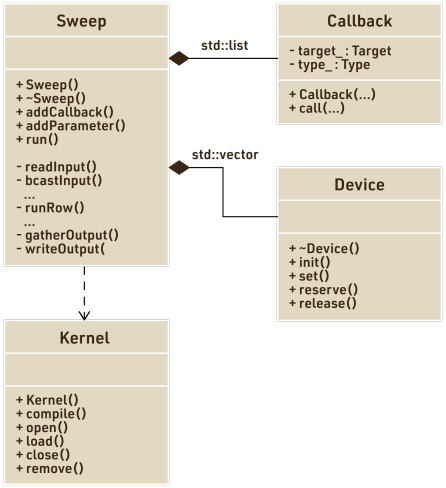

Figure 11: Basic structure of BONSAI classes.

The main classes and their functions are as follows:

Kernel provides the main functionality related to the GPU kernel, such as compilation to a dynamic library and opening, loading, and closing of the dynamic library. It also stores the file system information, such as the path to the kernel source file, the location of the scratch folder, etc. One object of the `Kernel` class represents the kernel with one set of parameters (one record of the input CSV file).

Callback stores the main information about user-defined callbacks, such as their types (`Kernel`, `Test`, `Cleanup`, `Other`) and targets (`Host`, `Device`). One object of the `Callback` class represent one stage in the processing pipeline, like the one in Figure 5.

Device implements handling of devices and stores information such as the device number and a unique stream. It also provides the functionality of reserving a device for the duration of processing one record. One object of the `Device` class represents one GPU.

Sweep is the main class and contains the bulk of the implementation. The public methods allow the user to add callbacks and user-defined output parameters. It contains a lengthy constructor allowing the user to pass all the necessary setup options (Figure 4). Its private methods implement the file IO described in Section 5.3, and the parallel scheduling system described in Section 5.4.

Considerable attention has been paid to the software engineering, resulting in a fairly compact implementation. The code relies on C++ facilities for its file IO operations (`<fstream>`, `<iostream>`, `<sstream>`) and on standard library containers for its data structures (`<list>`, `<map>`, `<set>`, `<string>`, `<vector>`). It also uses C++ regular expression capabilities (`<regex>`). The code falls back on C facilities where appropriate, e.g., when interacting with the operating system (OS) through calls to the Portable Operating System Interface (POSIX). The implementation is fairly portable due to the use of MPI for messaging and OpenMP for multithreading. The only non-portable aspect of the code is the use of NVIDIA CUDA. Comments and Doxygen[10] sections were used extensively to make the code readable.

6 EXPERIMENTAL RESULTS

6.1 Kernel

While in the previous sections we used the simplistic `gemm` kernel from the CUDA Programming Guide, here we are using a highly parametrized `gemm` kernel developed in the course of our past autotuning efforts [2, 9, 10]. It is based on a fairly standard approach, where values of C are accumulated in registers, while values of A and B are streamed through shared memory in thin stripes. Figure 12 shows all the blocking factors of the implementation.

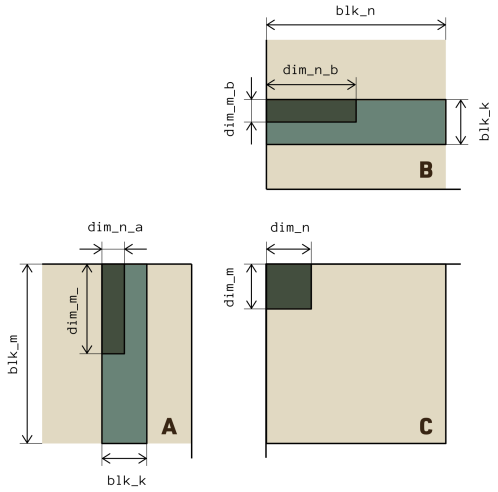

Figure 12: Blocking factors of the `gemm` implementation.

Each thread block is of size `dim_m` × `dim_n` and computes a block of C of size `blk_m` × `blk_n`. A is streamed through shared memory in stripes of size `blk_m` × `blk_k` and B is streamed in stripes of size `blk_k` × `blk_n`. For the lack of space, we refer the readers to the literature for more details [2, 9, 10].

[10] http://www.doxygen.nl

6.2 Environment

The hardware is a cluster of four nodes, each containing two 10-core Intel Xeon E5-2650 v3 (Haswell) CPUs, 4 NVIDIA GeForce GTX 1060 6GB (Pascal) GPUs, and a 56G InfiniBand (IB) FDR adapter. The nodes are connected using an SB7700 InfiniBand EDR 100G switch. The code was built using GCC 8.3.0, CUDA 10.1, and OpenMPI 4.0.0.

6.3 Search Space

The search space was generated using the LANguage for Autotuning Infrastructure (LANAI) [10]. Figures 13 and 14 show the complete LANAI input file used in the experiments. The file is slightly simplified compared to the file for an exhaustive gemm sweep. Here we are only tuning gemm in single precision and only for the case where A and B are not transposed. We also made the assumptions that `dim_m_a = dim_m_b = dim_m` and `dim_n_a = dim_n_b = dim_n`, i.e., the same shape of the thread block is used for access to A, B, and C. We are also using the default settings for the `cudaFuncCache` and `cudaSharedMemConfig` settings.

The LANAI file starts with a set of iterators, mostly corresponding to the blocking factors in Figure 12. The search space also includes two variants of the implementation: a simpler one based on the scalar type `float`, and a more complex one based on the vector type `float4`. Also, each of the input matrices, A and B, can be read using standard memory reads or using texture reads.

The LANAI file contains three user-defined heuristic thresholds, two of which—`min_threads_per_sm` and `min_fmas_per_load`—we use to control the size of the tuning sweep:

- `min_threads_per_sm` sets the lower limit on the number of threads per multiprocessor; e.g., forces minimum occupancy. We use the values of 512, and 256 for our sweeps. For the Pascal architecture, which has 2048 cores per multiprocessor, this translates to 25% and 12.5% occupancy.

```
from lanai import *
from device_constants.cuda_constants import *
from device_constants.GeForce_GTX_1060_6GB import *

max_threads_dim_x   = maxThreadsDim[0]                    # 1024
max_threads_dim_y   = maxThreadsDim[1]                    # 1024
max_regs_per_thread = MaxRegistersPerThread[major][minor]   #  255
#-------------------------------------------------------------
# iterators
dim_m = range(1, max_threads_dim_x+1)
dim_n = range(1, max_threads_dim_y+1)

@iterator
def blk_m(dim_m):
    return range(dim_m, max_threads_dim_x+1, dim_m)

@iterator
def blk_n(dim_n):
    return range(dim_n, max_threads_dim_y+1, dim_n)

blk_k = range(1, min(max_threads_dim_x, max_threads_dim_y)+1)
dim_vec = range(1, 5, 3)   # returns 1 for float and 4 for float4

@iterator                  # indicates if vector types are used
def vec_mul(dim_vec):      # in the main multiplication loop
    if dim_vec == 1:
        return range(0, 1) # returns 0 if the type is float
    else:
        return range(0, 2) # returns 0 and 1 if the type is float4

tex_a = range(0, 2)    # indicates if texture reads are used for A
tex_b = range(0, 2)    # indicates if texture reads are used for B
#-------------------------------------------------------------
# derived variables
threads_per_block = dim_m * dim_n

thr_m = blk_m / dim_m   # each thread computes thr_m x thr_n
thr_n = blk_n / dim_n   # part of C

float_size = 4                      # sizeof(float)
regs_per_thread = thr_m * thr_n     # registers required to store C

regs_per_block = regs_per_thread * threads_per_block
max_blocks_by_regs  = regsPerMultiprocessor / regs_per_block
max_threads_by_regs = max_blocks_by_regs * threads_per_block

shmem_per_block = blk_k * (blk_m + blk_n) * float_size
max_blocks_by_shmem  = sharedMemPerMultiprocessor / shmem_per_block
max_threads_by_shmem = max_blocks_by_shmem * threads_per_block

# number of active blocks per streaming multiprocessor (SM)
blocks_per_sm = min(max_blocks_by_regs, max_blocks_by_shmem)
# number of active threads per streaming multiprocessor (SM)
threads_per_sm = blocks_per_sm * threads_per_block

loads_per_block = (blk_m + blk_n) * blk_k / dim_vec
fmas_per_thread = thr_m * thr_n * blk_k
fmas_per_block  = fmas_per_thread * threads_per_block
```

Figure 13: LANAI search space definition - part 1.

```
#-------------------------------------------------------------
# user-defined heuristic thresholds

# use at least 512 threads per streaming multiprocessor (SM)
min_threads_per_sm = 512

# use at least 2 blocks per streaming multiprocessor (SM)
min_blocks_per_sm = 2

# have at least 64 FMA instructions per one load instruction
min_fmas_per_load = 64
#-------------------------------------------------------------
# hard constraints
@condition                  # too many threads per block
def over_max_threads(threads_per_block):
    return threads_per_block > maxThreadsPerBlock

@condition                  # too many registers per thread
def over_max_regs_per_thread(regs_per_thread):
    return regs_per_thread > max_regs_per_thread

@condition                  # too many registers per block
def over_max_regs_per_block(regs_per_block):
    return regs_per_block > regsPerBlock

@condition                  # exceeding the size of shared memory
def over_max_shmem(shmem_per_block):
    return shmem_per_block > sharedMemPerBlock
#-------------------------------------------------------------
# implementation correctness violations
@condition
def cant_reshape_a(blk_m, blk_k, dim_m, dim_n):
    return ((blk_m % (dim_m*dim_vec) != 0) or (blk_k % dim_n != 0))

@condition
def cant_reshape_b(blk_k, blk_n, dim_m, dim_n):
    return ((blk_k % (dim_m*dim_vec) != 0) or (blk_n % dim_n != 0))
#-------------------------------------------------------------
# soft heuristics
@condition                  # blocks not divisible by warp size
def partial_warps(threads_per_block):
    return threads_per_block % warpSize != 0

@condition                  # not enough blocks per multiprocessor
def min_block(blocks_per_sm):
    return blocks_per_sm < min_blocks_per_sm

@condition                  # too low occupancy
def low_occupancy(threads_per_sm):
    return threads_per_sm < min_threads_per_sm

@condition                  # too few FMAs per load
def low_fmas(loads_per_block, fmas_per_block):
    return fmas_per_block < min_fmas_per_load * loads_per_block
```

Figure 14: LANAI search space definition - part 2.

Table 1: Results of `gemm` performance tuning sweeps.

sweep size	min threads per SM	min FMAs per load	sweep run time [dd:hh:mm:ss]	approx. serial time [dd:hh:mm:ss]	approx. speedup	approx. efficiency
5,980	512	64	1:17:20	18:11:03	14.1	88%
20,428	512	32	3:11:09	43:26:19	13.6	85%
38,924	512	16	6:29:29	90:00:42	13.9	87%
63,532	256	32	1:09:07:38	20:09:14:45	14.8	92%

- `min_blocks_per_sm` sets the lower limit on the number of thread blocks per multiprocessor. We use the value of 2. This is because the multiprocessor has 96K of shared memory, while one thread block can only use 48K. So, at least two blocks are required to fully utilize the shared memory.
- `min_fmas_per_load` sets the minimum number of fused multiply–add (FMA) instructions per one load instruction, forcing a certain level of computing intensity. We use the values of 64, 32, and 16 for our sweeps.

The LANAI file ends with pruning conditions for eliminating undesirable cases. Hard constraints eliminate violations of hardware limits. Implementation correctness checks protect against unimplemented corner cases. Soft heuristics enforce performance guidelines, and take into account user-defined thresholds.

6.4 Results

We set up four performance tuning sweeps of different sizes by changing the values of the parameters `min_threads_per_sm` and `min_fmas_per_load` (lines 66 and 72 in Figure 14 respectively). Table 1 shows the results. All runs were done with $m = n = k = 10,000$, i.e., all matrices were of size $10,000 \times 10,000$. Each kernel was run five times. Work was dispatched in chunks of size 100.

We approximated the time of serial execution by summing up the `Time` column in the output CSV files and multiplying by the number of iterations, which was five in this particular case. This really is a lower bound of the serial execution time. First, the fastest run of each kernel is used as an approximation for the slower four. Second, the time of all CPU tasks is completely ignored This includes the time to compile the kernel, initialize the data, and check the results for errors. Nevertheless, the approximate parallel efficiency is between 85% and 92%.

Although actually finding the fastest `gemm` kernel was not the true objective of the exercise, there is no harm in reporting it. The top performer achieved 3,180 GFLOPS compared to the 3,875 GFLOPS for cuBLAS. I.e., our kernel achieved 82% of cuBLAS performance, which is fairly good for a kernel compiled from C++ source code, as opposed to cuBLAS `gemm`, which is implemented in assembly. The values of tuning parameters were: `dim_m = dim_n = 16`, `blk_m = blk_n = 128`, `blk_k = 32`, no vectorization, no texture reads.

The real goal here was to show the impact of using parallel resources for autotuning runs. Table 1 shows how using a very modest number of GPUs (16) can dramatically reduce the run time of a tuning sweep. Basically, days are reduced to hours. The largest run of size 63,532 took 1 day and 9 hours to execute in parallel, while the estimated serial time is 20 days and 9 hours. This opens up all kinds of new opportunities when targeting systems with thousands of GPUs like Summit and Sierra.

7 FUTURE PLANS

We believe that the initial implementation, presented in this article, provides fairly powerful and unique capabilities. At the same time, there is great potential for both improving the performance of the system and extending its functionality.

Here are some of the main ideas for performance improvements:

- Right now, the kernel is recompiled for each record in the input CSV file. Clearly, recompilation is not necessary if only runtime parameters change from record to record. The records can easily be sorted to group records with identical values of compile-time parameters, and skip recompilation. This would be very beneficial for kernels sensitive to the input data, which need to be bench-tested with the same set of compile-time parameters and different datasets.
- Currently, scheduling of local tasks is somewhat suboptimal due to the fact that a GPU is requested right after compilation and held for the rest of the pass. This way the device may be reserved before any work is sent to the GPU. Alternatively, the device could be requested before the first `Device` task and released after the last `Device` task. This would allow for more `Host` tasks to execute on the CPU without holding the GPU.
- Normally, one kernel, corresponding to one input record, is run many times, in order to produce a statistically meaningful measure of the execution time. At the same time, if a kernel shows inferior performance in one or two runs, say, 10× slower than the best discovered so far, then the remaining iterations can be skipped. Because the system relies on a centralized server for work dispatch, we can easily keep track of the top performance (minimum execution time). The clients can report it in their work requests. The server can send updates in the replies.
- There could also be a user-defined timeout to protect against very long compilation times, although this is questionable, as excessive compilation time can still produce a fast kernel.
- NVIDIA provides the NVRTC tool for runtime compilation.[11] NVRTC accepts source code in character string form and creates handles that can be used to obtain the parallel thread execution (PTX) code. The PTX string generated by NVRTC can then be loaded and linked. The use of NVRTC could lead to shorter compilation times, relieve the stress from the

[11] https://docs.nvidia.com/cuda/nvrtc/

file system, and solve the potential problem of the NVCC compiler not being accessible from the production nodes of a supercomputer (only the head nodes). It is still important to have the option to go through the file system in situations where compilation is more complex than just a simple NVCC invocation. One good example is the use of a more powerful preprocessor, like the pyexpander.[12]

Here are some of the main ideas for functionality extensions:

- Currently, the system detects three types of failures: the failure to compile, the failure to launch, and the failure of the user-defined test. It is also possible that the kernel compiles and launches, but then crashes during execution. We would like to detect execution failures and report them like the other failures.
- Right now, the system only measures the execution time. We argue that collecting information from hardware performance counters could provide a treasure trove of information for data analysis and machine learning. Ideally, the user could specify which events or metrics to collect and BONSAI would collect them automatically and return in the output CSV file.
- Currently, we do not have a good measure of the utilization/speedup, i.e., a comparison to a serial run (one CPU thread, one GPU device). It makes little sense to set up a run like that just to get that number. On the other hand, BONSAI can produce a good approximation by collecting the times of all CPU tasks and all GPU tasks launched in a parallel run.
- Right now, we are only returning the best performance for each record (minimum time). We could also return other measures, such as: maximum, average, median, standard deviation. This could easily be configurable by the user.
- We could also make the number of iterations a function of, e.g., the standard deviation. I.e., iterate until the standard deviation drops below a certain threshold (or some hard limit on the number of iterations is reached).

Finally, from the standpoint of working with a supercomputer or a large cluster, it would be helpful to have the ability to execute a range of records from the input file. Such environments rely on job schedulers, like the Portable Batch System (PBS) or Slurm, and jobs are submitted through batch queues which impose time limits. A truly large sweep may have to be cut into smaller batches.

8 CONCLUSIONS

In this article we built the case for massively parallel automatic software tuning by arguing that there is both the need and the opportunity. We described an implementation of a specialized system for deploying large tuning sweeps of GPU kernels to large supercomputer/cluster installations. We tried to cover all the avenues of the system: support for different types of parameters, detection of different types of failures, distributed-memory dispatch, and node-level dynamic scheduling. We believe that the system offers powerful and unique capabilities and can make a profound impact on the field of automatic performance tuning. We also outlined the most important direction for future developments.

[12] http://pyexpander.sourceforge.net

SOFTWARE

The BONSAI software is freely available at https://bitbucket.org/icl/bonsai. It is distributed under the modified BSD license, imposing only minimal restrictions on its use and redistribution. For assistance with BONSAI, email <bonsai-user@icl.utk.edu>.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grant No. 1642441 (SI2-SSE: BONSAI: An Open Software Infrastructure for Parallel Autotuning of Computational Kernels).

REFERENCES

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 303–316.

[2] Hartwig Anzt, Blake Haugen, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. 2015. Experiences in autotuning matrix multiplication for energy minimization on GPUs. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 5096–5113.

[3] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th international conference on Supercomputing*. ACM, 340–347.

[4] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 2011. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 676–687.

[5] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.

[6] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, Vol. 3. IEEE, 1381–1384.

[7] Shoaib A Kamil. 2013. *Productive high performance parallel programming with auto-tuned domain-specific embedded languages*. Ph.D. Dissertation. Electrical Engineering and Computer Sciences, University of California at Berkeley.

[8] Takahiro Katagiri, Kenji Kise, Hiroaki Honda, and Toshitsugu Yuba. 2003. Fiber: A generalized framework for auto-tuning software. In *International Symposium on High Performance Computing*. Springer, 146–159.

[9] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. 2012. Autotuning GEMM kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems* 23, 11 (2012), 2045–2057.

[10] Piotr Luszczek, Mark Gates, Jakub Kurzak, Anthony Danalis, and Jack Dongarra. 2016. Search space generation and pruning system for autotuners. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 1545–1554.

[11] Kengo Nakajima, Masaki Satoh, Takashi Furumura, Hiroshi Okuda, Takeshi Iwashita, Hide Sakaguchi, Takahiro Katagiri, Masaharu Matsumoto, Satoshi Ohshima, Hideyuki Jitsumoto, et al. 2016. ppOpen-HPC: open source infrastructure for development and execution of large-scale scientific applications on post-peta-scale supercomputers with automatic tuning (AT). In *Optimization in the Real World*. Springer, 15–35.

[12] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2010. An improved MAGMA GEMM for Fermi graphics processing units. *The International Journal of High Performance Computing Applications* 24, 4 (2010), 511–515.

[13] Markus Püschel, José MF Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W Johnson. 2004. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *The International Journal of High Performance Computing Applications* 18, 1 (2004), 21–45.

[14] David E Tanner. 2018. Tensile: Auto-Tuning GEMM GPU Assembly for All Problem Sizes. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1066–1075.

[15] Cristian Ţăpuş, I-Hsin Chung, Jeffrey K Hollingsworth, et al. 2002. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1–11.

[16] Richard Vuduc, James W Demmel, and Katherine A Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, Vol. 16. IOP Publishing, 521.

[17] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1-2 (2001), 3–35.