Multi-Target Autotuning for Accelerators

Nicholas A. Chaimov
nchaimov@cs.uoregon.edu
University of Oregon

March 18, 2014

Abstract

Considerable computational resources are available on GPUs and other accelerator devices, the use of which can offer dramatic increases in performance over traditional CPUs. However, programming such devices can be difficult, especially given considerable architectural differences between different models of accelerators. The OpenCL language provides portable code, but does not provide performance-portability: code which is optimized to run well on one device will run poorly on others. As a solution to this problem, we have developed OrCL, an autotuning system which generates OpenCL kernels from a subset of C, searching a space of variant implementations for the best-performing version for a particular problem and device. We instrument the resulting implementations in order to measure the performance of variants across the search space for NVIDIA GPUs, AMD GPUs, and Intel Xeon Phi accelerators for a set of numerical kernels used in sparse linear system solvers, as well as computations from a radiation transport simulation code.

1 Introduction

Achieving optimal performance of numerical code requires optimizations which vary with the input code, the environment in which the code will be executed, and the input dataset. Performing these optimizations manually is time-consuming and is outside the area of expertise of many domain scientists. Carrying out an optimization for one particular computer architecture renders the code more difficult to read and less portable to other architectures. This is a problem in a heterogenous environment, in which a single application may run across multiple architecture types, as well as in cloud computing, where the architectures on which the code may ultimately run are not necessarily known until they are run [19]. *Empirical autotuning* is the process of automatically generating code variants to which various transformations have been applied and measuring their performance, searching for the set of transformations which yields the greatest performance improvement. Through autotuning, general-purpose code can be automatically adapted to new execution environments and input datasets.

We have previously [9] worked to integrate the TAU Performance System [34] with existing autotuning frameworks. The first such framework [37] combines CHiLL [10], a polyhedral-model code variant generator, and Active Harmony [38], a parallel search engine which directs the generation of variants by CHiLL. The second is Orio [20], an annotation-based combination variant generator and search engine. In the integrated systems, TAU is used to instrument generated code variants [26], with performance profiles being stored in the TAUdb performance database. Profiles are annotated with a description of the transformations applied to produce the tested variant and metadata describing the execution environment and properties of the input data. We then use decision tree learning over performance data gathered over multiple autotuning sessions in different environments, generating a classifier which can be used at runtime to select from among a library of variants a variant expected to perform well.

The search space of possible transformation parameters is very large [3], so autotuning can be very time consuming. If autotuning is to be widely adopted, it should not take too much time to carry out the process. The time required to carry out autotuning is a function of the number of evaluations needed during search and the time to carry out each evaluation. Since the objective function essentially *is* the amount of time needed to carry out evaluation, the autotuning process will, given the same number of evaluations, nonetheless complete more quickly if fewer of those evaluations are of poorly-performing variants.

Evaluation of the performance of different search algorithms [2] over a set of linear algebra and stencil kernels [4] shows that the selection of search algorithm has an effect on both the performance of the best variant found during search and on the amount of time needed to carry out the search (see Figure 1). Which search algorithm results in finding the best variant, and which completes the fastest, varies with the particular code being autotuned.

That work shows variation in the search space across codes, but did not evaluate variation across architectures or languages. In order to more enable a more thorough evaluation of the effect of architectural features on the performance of codes, we have implemented OpenCL code generation and OpenCL-specific optimizations in the Orio autotuning framework. We then hold input code constant while varying the generated language (CUDA vs. OpenCL) and, for OpenCL, the target architectures (two AMD Radeon GPUs, three NVIDIA GPUs, and the Intel Xeon Phi) for several BLAS kernels and for kernels from a radiation transport simulation code. We observe that the best code variant founds varies with the input code, the input data problem size, and the architecture in a complex manner, suggesting that a machine learning approach to the problem would be valuable for multi-architecture autotuning.

2 Related Work

Autotuning systems are of three types: *library-based*, *language-based*, and *compiler-based* [7]. Library-based autotuning systems perform benchmarking at compile time to select good-performing variants based on the environment in which they are be-

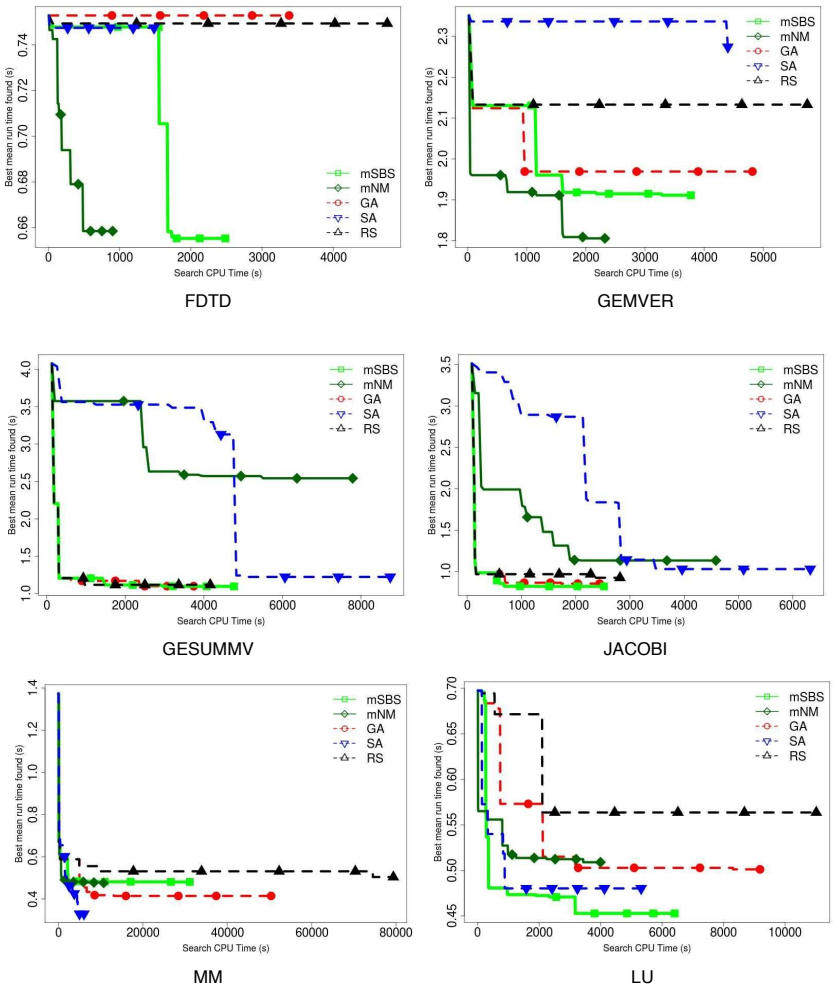

Figure 1: Performance of different search algorithms during autotuning of different codes. Adapted from Figures 1, 2, and 3 of [2].

ing installed. The linear algebra library ATLAS [39] (Automatically Tuned Linear Algebra Software) is a widely-used example of library-based autotuning. An alternative is to perform benchmarking at runtime, once parameters of the input data are known. The Fourier Transform library FFTW [18] does this. Language-based autotuning systems, such as PetaBricks [1], use special-purpose programming languages which allow the programmer to specify alternate implementations of a computation. Compiler-based autotuning systems automatically generate alternative implementations at compile time from code written in existing languages. CHiLL [10, 37] (with an external search engine such as Active Harmony [37]) and Orio [20] are examples of compiler-based autotuning systems. These systems have some overlap with language-based systems, as these allow programmers to specify parameters to source-to-source transformations, which generate code variants.

There has been considerable interest in autotuning for GPUs and other accelerator devices. A library-based system has been built around MAGMA [31], a linear algebra library similar to ATLAS, but for accelerators. Library-based systems have been developed for dense-matrix multiplication routines [24], sparse-matrix multiplication routines [11], N-body simulations [13], molecular dynamics simulations [32], tensor contraction routines [35], and Fast Fourier Transform routines [14]. CUDA-CHiLL [33] and OrCUDA [28, 12] are compiler-based systems which generate CUDA code variants from a subset of C.

Most existing autotuning work has been done using the CUDA programming model, although an auto-tuning FFT implementation for OpenCL, MPFFT, has been developed [25]. Comparisons of CUDA and OpenCL implementations of kernels implementing the same computation have found that roughly equivalent performance can be obtained, but that different optimizations are required to obtain equivalent performance [16]. As a result, systems that directly translate optimized CUDA kernels into OpenCL kernels, such as CU2CL [29], will not necessarily provide good performance, and Du et al. [15] have suggested autotuning as a means of achieving performance portability for OpenCL kernels.

3 OpenCL Code Generation

The capabilities of execution units and the sizes and performance characteristics of the memories vary among devices, especially between generations of devices and between devices from different vendors. While OpenCL provides portability in the sense that kernels originally designed for one device will run on another, they are not *performance-portable*: optimizations that yield good performance on one device will often not yield good performance on another device. Given this constraint, automatic performance tuning can be used to search for variants with good performance.

To accomplish this, we have extended the Orio autotuning framework with a transformation module, OrCL, which takes as input a kernel specified in a subset of C (also referred to as the "loop language") and a set of transformation parameters and outputs OpenCL device and host code, much as the OrCuda module described in [28] does for CUDA. OrCuda and OrCL accept the same kernel specifications

and, with some exceptions, the same transformation specifications, allowing for code to be generated for either CUDA or OpenCL as desired.

When using OrCL, the user identifies loops in his or her application which are targets for execution on an accelerator device. This loop is then wrapped with a tuning specification by placing annotations as comments before and after the original loop in the code, so that the original code can still be executed normally. When Orio is invoked on the code, variants are generated and tested based on the specification. An example of an OrCL annotation is shown in Figure 3. The tuning specification consists of several regions: the `performance_params` section, describing the parameter values which make up the search space for autotuning; the `build` section, which describes how generated variants can be compiled; the `input_params` section, which describes properties of the inputs against which generated variants are tested; the `input_vars` section, which specifies the values of the inputs; and the `performance_counter` and `performance_test_code` sections, which describe how performance measurements of the generated variants are to be made. The tuning specification is followed by a transformation statement, which makes use of the parameter values described in the tuning specification.

1) *workGroups:* The number of OpenCL work groups to use. This, multiplied by the number of work items per work group, gives the overall number of threads that make up a kernel invocation, the *global work size*. Where this number is smaller than the size of the data to be processed, kernels are generated such that each work item processes more than one input. The requested number of work groups are then scheduled across the compute units of the device by the OpenCL runtime.

2) *workItemsPerGroup:* The number of work items (threads) that make up each work group; this controls the *local work size*. Each device has a maximum number of work items per group, which can be queried on the host. Using the maximum number of work items makes use of all the computational resources within a workgroup; however, because the entire work group shares a pool of local memory, increasing

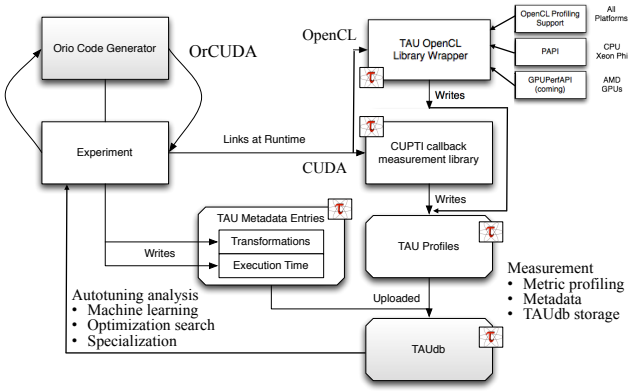

Figure 2: Data flow between Orio, TAU, and TAUdb.

```
void VecAXPY(int n, double a, double *x, double *y) {

  register int i;
  /*@ begin PerfTuning(
        def performance_params {
          param WI[] = [32,64,128,256];
          param WG[] = [4,8,16,32,64,128];
          param CB[] = [True, False];
          param SH[] = [True, False];
          param UI[] = range(1,4);
          param VH[] = [0,2,4];
          param CL[] = ['','-cl-fast-relaxed-math'];
        }
        def build {
          arg build_command = 'gcc -O3 -lOpenCL';
        }
        def input_params {
          param N[] = [100000,1000000];
        }
        def input_vars {
          decl double a = random;
          decl static double x[N] = random;
          decl static double y[N] = 0;
        }
        def performance_counter {
          arg method = 'basic timer';
          arg repetitions = 5;
        }
        def performance_test_code {
          arg skeleton_code_file = 'tau_skeleton.c';
        }
  ) @*/

  int n=N;

  /*@ begin Loop(transform OpenCL(workGroups=WG, workItemsPerGroup=WI, sizeHint=SH, vecHint=VH,
        cacheBlocks=CB, unrollInner=UI, clFlags=CL, device=1, platform=0)

  for (i=0; i<=n-1; i++)
    y[i]+=a*x[i];

  ) @*/

  for (i=0; i<=n-1; i++)
    y[i]+=a*x[i];

  /*@ end @*/
  /*@ end @*/
}
```

Figure 3: Annotated vecAXPY kernel for OpenCL generation.

the number of work items decreases the available memory per work item, which can result in the spilling of data into global memory, which is much slower than local memory. The optimum number of work items per group is thus dependent on the memory usage of the work items.

3) sizeHint: OpenCL provides a pair of function attributes which provide hints to the compiler about the expected local work size. The `work_group_size_hint` attribute allows the compiler to make optimizations that improve performance when the local work size is equal to the hinted size but might degrade performance otherwise. The `reqd_work_group_size` attribute makes it an error to invoke the kernel with any work group size other than the hinted size, allowing the compiler to make optimizations that would yield incorrect results for other sizes. OpenCL compilers,

```
const char* orcl_kernel_source="#pragma␣OPENCL␣EXTENSION␣cl_khr_fp64␣:␣enable\n"
"__kernel␣__attribute__((vec_type_hint(double2),work_group_size_hint(64,1,1),
      reqd_work_group_size(64,1,1)))␣void␣orcl_kernel(const␣int␣n,␣double␣a,␣__global␣double*␣y,
      ␣__global␣double*␣x)␣{\n"
"␣␣const␣size_t␣tid=get_global_id(0);\n"
"␣␣const␣size_t␣gsize=get_global_size(0);\n"
"␣␣#pragma␣unroll␣2\n"
"␣␣for␣(int␣i=tid;␣i<=n-1;␣i+=gsize)␣{\n"
"␣␣␣␣y[i]=y[i]+a*x[i];\n"
"␣␣}\n"
"}\n"
"";
```

Figure 4: Example of a generated OpenCL vecAXPY kernel.

however, are not required to make use of these hints. The `sizeHint` transformation parameter specifies whether these attributes should be applied to generated kernels.

4) *vecHint:* OpenCL provides another function attribute which provides information to the compiler as to how the function should be autovectorized. The `vec_type_hint` attribute informs the compiler of the width of the data consumed by the kernel, which the compiler can use to merge or split work items to enable better use of vector operations. As with the size hints, the compiler is not required to make use of the hint.

5) *cacheBlocks:* A parameter specifying whether work items should copy input data located in global memory into local memory before operating on it. This can improve performance, especially when global memory is not cached, at the expense of increasing the memory consumption of each work item.

6) *unrollInner:* A parameter specifying whether to provide a hint to the compiler as to an unroll factor for the innermost loop in the kernel. This is done by inserting a pragma before the loop in the kernel source code. This is an OpenCL extension which is not necessarily supported by all implementations.

7) *clFlags:* Flags provided to the OpenCL compiler to control optimization. These can be `-cl-mad-enable` to enable fused multiply-add instructions; `-cl-no-signed-zeros` to ignore the signedness of zeroes; `-cl-unsafe-math-optimizations` to assume all arguments to floating-point arithmetic operators are valid; `-cl-finite-math-only` to assume that all arguments to floating-point arithmetic operators are finite numbers; and `-cl-fast-relaxed-math`, which combines the effects of the previous two flags.

8) *device* and *platform*: In a system with more than one OpenCL platform and/or device available, a parameter specifying which of these to use. If used as a tuning parameter, autotuning will be attempted with each of the platforms and devices specified and the device providing the highest performance will be used in the generated code.

Some optimizations available in the CUDA code generator are not yet available in the OpenCL generator. The *streamCount* parameter to the CUDA code generator

```
cl_int orcl_status;
cl_uint orcl_num_platforms;
cl_platform_id * orcl_platforms;
orcl_status=clGetPlatformIDs(0,NULL,&orcl_num_platforms);
/*get platforms*/
orcl_platforms=malloc(orcl_num_platforms*sizeof(cl_platform_id));
orcl_status=clGetPlatformIDs(orcl_num_platforms,orcl_platforms,NULL);
/*get number of devices for chosen platform*/
cl_uint orcl_num_devices;
cl_device_id * orcl_devices; orcl_status=clGetDeviceIDs(orcl_platforms[0],CL_DEVICE_TYPE_ALL,0,
    NULL,&orcl_num_devices);
if (orcl_status!=CL_SUCCESS) {
fprintf(stderr,"OpenCL Error: %d in %s \\n",orcl_status,"clGetDeviceIDs for number");
exit(EXIT_FAILURE);
}
/*get devices for chosen platform*/
orcl_devices=malloc(orcl_num_devices*sizeof(cl_device_id)); orcl_status=clGetDeviceIDs(
    orcl_platforms[0],CL_DEVICE_TYPE_ALL,orcl_num_devices,orcl_devices,NULL);
/*create OpenCL context*/
cl_context orcl_context;
orcl_context=clCreateContext(NULL,orcl_num_devices,orcl_devices,NULL,NULL,&orcl_status);
/*create OpenCL command queue*/
cl_command_queue orcl_command_queue;
orcl_command_queue=clCreateCommandQueue(orcl_context,orcl_devices[1],CL_QUEUE_PROFILING_ENABLE,&
    orcl_status);
clFinish(orcl_command_queue);
/*declare variables*/
cl_mem dev_y, dev_x;
int nthreads=64;
/*calculate device dimensions*/
size_t orcl_global_work_size[1], orcl_local_work_size[1];
orcl_global_work_size[0]=4096;
orcl_local_work_size[0]=64;
/*allocate device memory*/
dev_y=clCreateBuffer(orcl_context,CL_MEM_READ_WRITE,N*sizeof(double),NULL,&orcl_status);
dev_x=clCreateBuffer(orcl_context,CL_MEM_READ_WRITE,N*sizeof(double),NULL,&orcl_status);
/*copy data from host to device*/
orcl_status=clEnqueueWriteBuffer(orcl_command_queue,dev_y,CL_FALSE,0,N*sizeof(double),y,0,0,NULL
    );    orcl_status=clEnqueueWriteBuffer(orcl_command_queue,dev_x,CL_FALSE,0,N*sizeof(double
    ),x,0,0,NULL);
/*compile kernel*/
cl_program orcl_kernel_program;
orcl_kernel_program=clCreateProgramWithSource(orcl_context,1,(const char **)&orcl_kernel_source,
    NULL,&orcl_status);
orcl_status=clBuildProgram(orcl_kernel_program,1,&orcl_devices[1],"-cl-fast-relaxed-math",NULL,
    NULL);
cl_kernel orcl_kernel_kernelobj;
orcl_kernel_kernelobj=clCreateKernel(orcl_kernel_program,"orcl_kernel",&orcl_status);
/*invoke device kernel*/
orcl_status=clSetKernelArg(orcl_kernel_kernelobj,0,sizeof(int),&n);
orcl_status=clSetKernelArg(orcl_kernel_kernelobj,1,sizeof(double),&a);
orcl_status=clSetKernelArg(orcl_kernel_kernelobj,2,sizeof(cl_mem),&dev_y);
orcl_status=clSetKernelArg(orcl_kernel_kernelobj,3,sizeof(cl_mem),&dev_x);
orcl_status=clEnqueueNDRangeKernel(orcl_command_queue,orcl_kernel_kernelobj,1,NULL,
    orcl_global_work_size,orcl_local_work_size,0,NULL,NULL);
/*copy data from device to host*/
orcl_status=clEnqueueReadBuffer(orcl_command_queue,dev_x,CL_TRUE,0,N*sizeof(double),x,0,NULL,
    NULL);
clFinish(orcl_command_queue);
```

Figure 5: Example of generated host code invoking the OpenCL vecAXPY kernel. Error checking and host memory management code have been removed for brevity.

splits the execution into multiple, overlapping transfers and kernel executions by using CUDA asynchronous streams. This could be implemented in OpenCL by using out-of-order command queues on devices which support that option, but this has not yet been done in OrCL. The *preferL1Size* parameter uses a feature of the CUDA API to choose how to apportion physical memory on the device into L1

cache and shared memory. This feature is not exposed in the OpenCL standard, nor is it exposed in any of the currently existing extensions.

3.1 Performance Measurement with TAU

For each requested combination of values of the above parameters, OrCL generates OpenCL device code (Figure 4) and host (Figure 5) code. In order to measure performance, the generated code is inserted into a skeleton which specifies how measurements are to be made. The skeleton code may be chosen from a library or the user may specify a custom skeleton. For integration with the TAU performance measurement system [34], we use a skeleton code which wraps the generated code in TAU instrumentation API calls. The time to execute the generated code can then be measured, along with requested hardware performance counters supported by TAU through PAPI [30]. In the case of CUDA or OpenCL code, the generated code is invoked through a TAU library wrapper, in which an instrumented version of the CUDA or OpenCL runtime library is used, which captures performance measurements for CUDA and OpenCL API calls before passing those calls on to the installed runtimes [27]. For CUDA, hardware performance counters can be sampled using CUPTI. For OpenCL, timings from OpenCL events are captured; however, hardware performance counters are not currently supported because OpenCL does not provide a standardized method for accessing performance counters, and NVIDIA's CUPTI does not support OpenCL.

For each variant tested, performance information gathered by TAU is stored into TAUdb, a database system for storing performance profiles and related metadata [22]. The performance data is annotated with metadata recording properties of the execution environment (such as the accelerator card used and the sizes of its memories), input data (such as the sizes of inputs), and optimizations applied (values chosen for each of the tunable parameters). This workflow is shown in Figure 2. Data stored in TAUdb can be analyzed using the *ParaProf* visualization system [8] and the *PerfExplorer* data mining framework [21, 23]. PerfExplorer analyses can be performed with a GUI or scripted by using a Python interface.

4 Evaluation

To evaluate OrCL, we ran a series of autotuning experiments on two AMD GPUs, three NVIDIA GPUs, and an Intel Xeon Phi. Specifications for the test hardware are

Table 1: Properties of OpenCL target platforms

Accelerator Device	Radeon 6970	Radeon 7970	GTX 480	Tesla C2075	Tesla K20C	Xeon Phi
OpenCL Version	1.2	1.2	1.1	1.1	1.1	1.2
Max Compute Units	24	32	15	14	13	204
Max Work Items	(256,256,256)	(256,256,256)	(1024,1024,64)	(1024,1024,64)	(1024,1024,64)	(1024,1024,1024)
Max Workgroup Size	256	256	1024	1024	1024	1024
Clock Frequency	880 MHz	1000 MHz	1401 MHz	1147 MHz	705 MHz	2000 MHz
Cache Size	None	16 KB	24 KB	224 KB	208 KB	None
Global Memory Size	1024 MB	2048 MB	1535 MB	5375 MB	4800 MB	2835 MB
Constant Buffer Size	64 KB	64 KB	64 KB	64 KB	64 KB	128 KB
Local Memory Size	32 KB	32 KB	48 KB	48 KB	48 KB	32 KB
Preferred Workgroup Size Multiple	64	64	32	32	32	16

given in Table 4. To verify that the OpenCL code generator generates correct code, we generated code from specifications of BLAS kernels previously used with the OrCUDA code generator and checked that the generated OpenCL kernels produce the same output. Generated variants were instrumented with TAU as described in Section 3.1.

The numerical kernels we autotuned are a subset of the kernels that typically consume a significant portion of the solution time of Newton-Krylov nonlinear solvers, which are commonly used in applications based on the solution of nonlinear partial differential algorithms (PDEs) discretized on a regular grid. Table 2 lists the linear algebra kernels we considered. The operation notation is as follows: A designates a matrix; $x, x_1, ..., x_n$, y, and w are vectors; and $\alpha, \alpha_1, ..., \alpha_n$ are scalars.

Figure 6 shows the execution times of the best variants for the matrix and vector kernels on vectors of size 10^6 for each architecture by an exhaustive search of the parameter space of the implemented optimizations, as well as the execution of of the best CUDA variant found for the K20c GPU. We normalize all times by the execution time for the equivalent ViennaCL [36] implementations using the default configurations. For each of the kernels, the CUDA variant produced the best performance, and the OpenCL kernels for the three NVIDIA GPUs produced the next-best performance. The best variants for two AMD GPUs and the Xeon Phi did not perform as well. We believe that the OpenCL kernels on NVIDIA produced slightly worse results than CUDA kernels because of the absence of the stream count and L1 cache size optimizations in the OpenCL code generator. Compared with the ViennaCL static compilation results, our initial autotuning results are up to 2.5 times faster and slower in only a couple of cases on the Tesla C2075.

We also autotuned the function and Jacobian computations of a PETSc-based [6, 5] application solving a 3-D solid fuel ignition (SFI) problem, defined as the following boundary value problem

$$-\nabla^2 u - \lambda e^u = 0 \text{ in } [0,1] \times [0,1] \times [0,1]$$
$$u = 0 \text{ on the boundary}$$

which is discretized by using a finite-difference approximation with a seven-point (star) stencil in order to obtain a nonlinear system of equations. The system is then solved by using PETSc's Newton-Krylov iterative solvers which rely on the kernels in Table 2 (and a few others) and invoke the application-specific code we refer to as EX14FF (function computation) and EX14FJ (Jacobian computation). The

Table 2: Kernel specifications.

Kernel	**Operation**
matVec	$y = Ax$
vecAXPY	$y = \alpha x + y$
vecMAXPY	$y = y + \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_n x_n$
vecScale	$w = \alpha w$
vecWAXPY	$w = y + \alpha x$

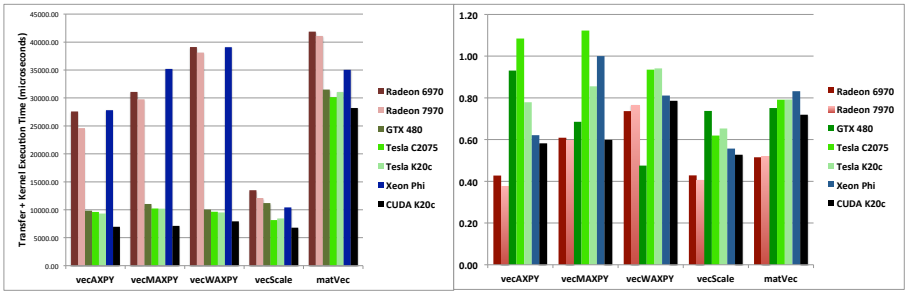

Figure 6: On left, absolute performance of the best-performing linear algebra kernel variant across devices. On right, performance normalized by time to run the corresponding ViennaCL version on the same device. The first six bars in each group correspond to code produced with OrCL, while the last bar is code produced with OrCUDA.

```
int m           = M;
int n           = N;
int p           = P;
int nrows       = m*n*p;
double hx       = 1.0/(m-1);
double hy       = 1.0/(n-1);
double hz       = 1.0/(p-1);
double sc       = hx*hy*hz*lambda;
double hxhzdhy  = hx*hz/hy;
double hyhzdhx  = hy*hz/hx;
double hxhydhz  = hx*hy/hz;

/*@ begin Loop(transform OpenCL(workGroups=WG, workItemsPerGroup=WI, clFlags=CFLAGS,
      unrollInner=UIF, sizeHint=SH, vecHint=VH, device=1)

for(i=0; i<=nrows-1; i++) {
  if (i<m*n || i>=nrows-m*n || i%(m*n)<m || i%(m*n)>=m*n-m || i%m==0 || i%m==m-1) {
    F[i] = X[i];
  } else {
    F[i] = (2*X[i] - X[i-1  ] - X[i+1  ])*hyhzdhx
         + (2*X[i] - X[i-m  ] - X[i+m  ])*hxhzdhy
         + (2*X[i] - X[i-m*n] - X[i+m*n])*hxhydhz
         - sc*exp(X[i]);
  }
}

) @*/
```

Figure 7: Input source code for the ex14FF kernel.

specifications for these kernels are shown in Figures 7 and 8, respectively. Most stencil-based computations can be similarly optimized by Orio, which is not limited to matrix algebra. We autotuned these kernels with four input sizes: 64^3, 75^3, 100^3, and 128^3 by using a Nelder-Mead-based algorithm for the search (unlike the exhaustive search used in the vector and matrix kernel tuning). These more complex kernels produced much more varied results from autotuning.

Figures 9 and 10 show the sorted execution times for all OpenCL code variants generated by the search for each of the kernels, sizes, and devices. The Radeon 6970 is omitted from results for all but the ex14FF 64^3 because that card did not

```
int m           = M;
int n           = N;
int p           = P;
int nrows       = m*n*p;
double hx       = 1.0/(m-1);
double hy       = 1.0/(n-1);
double hz       = 1.0/(p-1);
double sc       = hx*hy*hz*lambda;
double hxhzdhy  = hx*hz/hy;
double hyhzdhx  = hy*hz/hx;
double hxhydhz  = hx*hy/hz;

/*@ begin Loop(transform OpenCL(workGroups=WG, workItemsPerGroup=WI, clFlags=CFLAGS,
      unrollInner=UIF, sizeHint=SH, vecHint=VH, device=0)

for(i=0; i<=nrows-1; i++) {
  if (i<m*n || i>=nrows-m*n || i%(m*n)<m || i%(m*n)>=m*n-m || i%m==0 || i%m==m-1) {
    dia[i] = 1.0;
  } else {
    dia[i         ] = -hxhydhz;
    dia[i+  nrows] = -hxhzdhy;
    dia[i+2*nrows] = -hyhzdhx;
    dia[i+3*nrows] = 2.0*(hyhzdhx+hxhzdhy+hxhydhz) - sc*exp(x[i]);
    dia[i+4*nrows] = -hyhzdhx;
    dia[i+5*nrows] = -hxhzdhy;
    dia[i+6*nrows] = -hxhydhz;
  }
}
```

Figure 8: Input source code for the ex14FJ kernel.

Table 3: Parameter values for the best performing ex14FF and ex14FJ kernels across architectures. Result tuples are (workGroups, workItemsPerGroup, compileFlags, unrollInner, sizeHint, vecHint).

Accelerator Device	Radeon 6970	Radeon 7970	Xeon Phi
ex14FF 64^3	(64,64,'',1,False,0)	(64,32,'',2,False,0)	(64,64,'',2,True,2)
ex14FF 75^3	N/A	(16,32,'cl-fast-relaxed-math',4,False,4)	(32,128,'',2,False,2)
ex14FF 100^3	N/A	(32,32,'',4,True,0)	(128,64,'',1,True,0)
ex14FF 128^3	N/A	(32,64,'',4,True,4)	(16,256,'',1,False,0)
ex14FJ 64^3	N/A	(64,64,'cl-fast-relaxed-math',2,False,2)	(128,64,'cl-fast-relaxed-math',2,True,2)
ex14FJ 75^3	N/A	(64,256,'cl-fast-relaxed-math',1,False,0)	(64,256,'',2,False,2)
ex14FJ 100^3	N/A	(128,128,'',2,False,0)	(16,32,'',2,True,2)
ex14FJ 128^3	N/A	(32,128,'',2,False,2)	(32,64,'cl-fast-relaxed-math',4,False,2)

Accelerator Device	GTX 480	Tesla C2075	Tesla K20c
ex14FF 64^3	(16,64,'',1,False,2)	(64,64,'',2,True,0)	(64,128,'',1,False,0)
ex14FF 75^3	(128,64,'cl-fast-relaxed-math',2,True,2)	(16,128,'cl-fast-relaxed-math',2,False,0)	(32,32,'cl-fast-relaxed-math',2,True,0)
ex14FF 100^3	(128,128,'cl-fast-relaxed-math',4,True,0)	(64,128,'',1,True,2)	(64,64,'cl-fast-relaxed-math',4,False,2)
ex14FF 128^3	(32,32,'',2,False,2)	(64,128,'cl-fast-relaxed-math',1,True,0)	(32,32,'cl-fast-relaxed-math',False,0)
ex14FJ 64^3	(16,64,'',2,False,0)	(64,128,'',2,True,2)	(32,64,'',1,True,0)
ex14FJ 75^3	(64,256,'',2,False,2)	(64,128,'cl-fast-relaxed-math',2,True,2)	(128,64,'cl-fast-relaxed-math',1,True,0)
ex14FJ 100^3	(32,128,'',2,False,2)	(32,64,'cl-fast-relaxed-math',1,True)	(32,256,'',1,False,4)
ex14FJ 128^3	(32,128,'cl-fast-relaxed-math',4,False,2)	(32,128,'cl-fast-relaxed-math',2,True,2)	(32,64,'',4,False,0)

have enough memory to complete the other versions. For each device, the highest leftmost point represents the variant with the worst performance, while the lowest rightmost point represents the variant with the best performance. Each curve shows the distribution of points across the search space. The best-performing variant was found on different devices for different kernels and sizes: on the Xeon Phi for the ex14FF 64^3, ex14FF 128^3 and ex14FJ 128^3 kernels; on the Radeon 7970 for the ex14FF 75^3, ex14FF 100^3 and ex14FJ 100^3 kernels; and on the Tesla K20c for the ex14FJ 64^3 and ex14FJ 75^3 kernels.

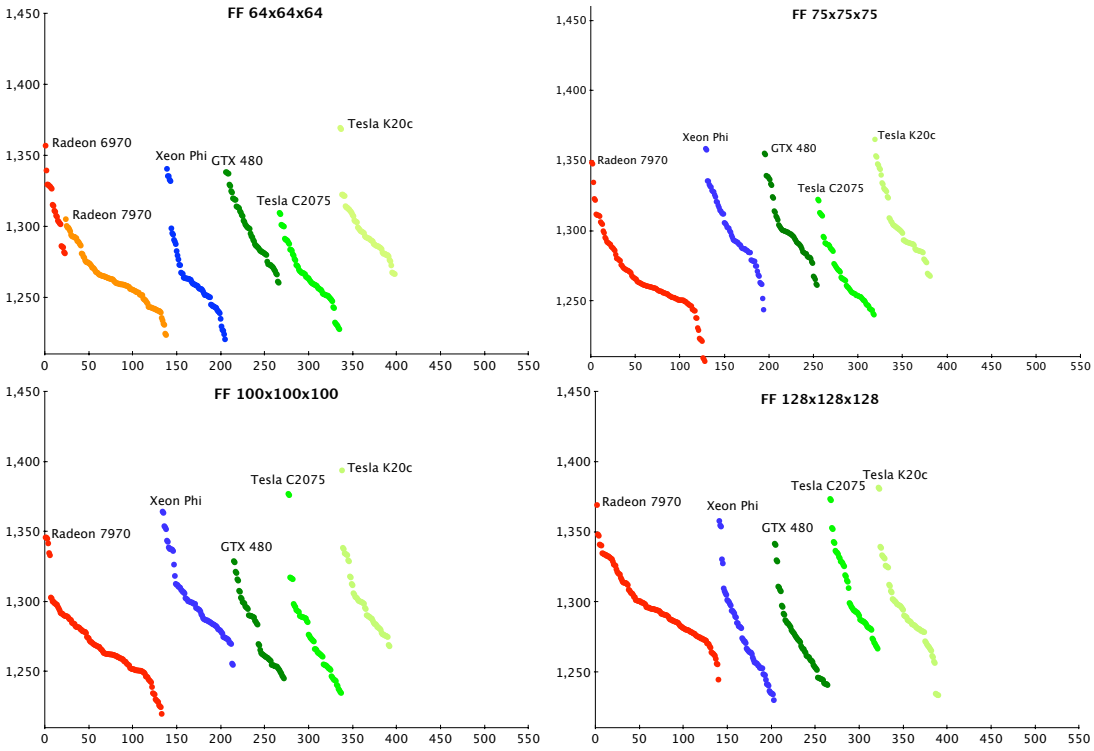

Figure 9: Performance (execution time in milliseconds) of evaluated variants for FormFunction3D kernels of four sizes.

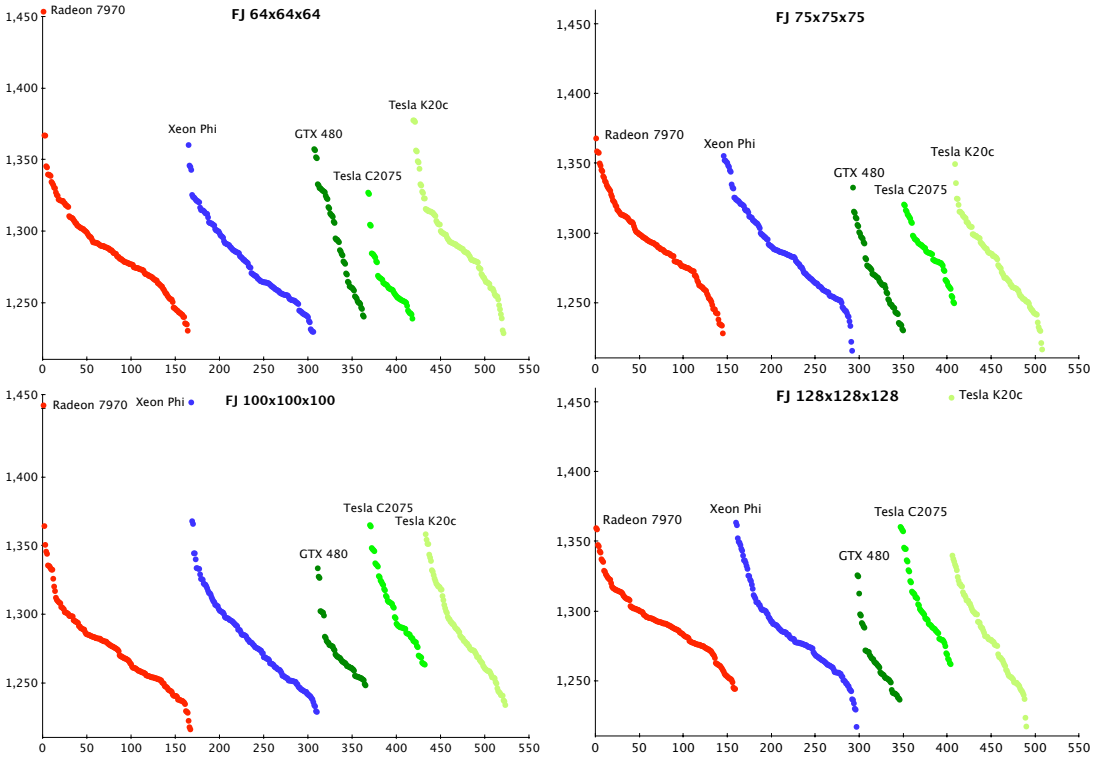

Figure 10: Performance (execution time in milliseconds) of evaluated variants for FormJacobian3D kernels of four sizes.

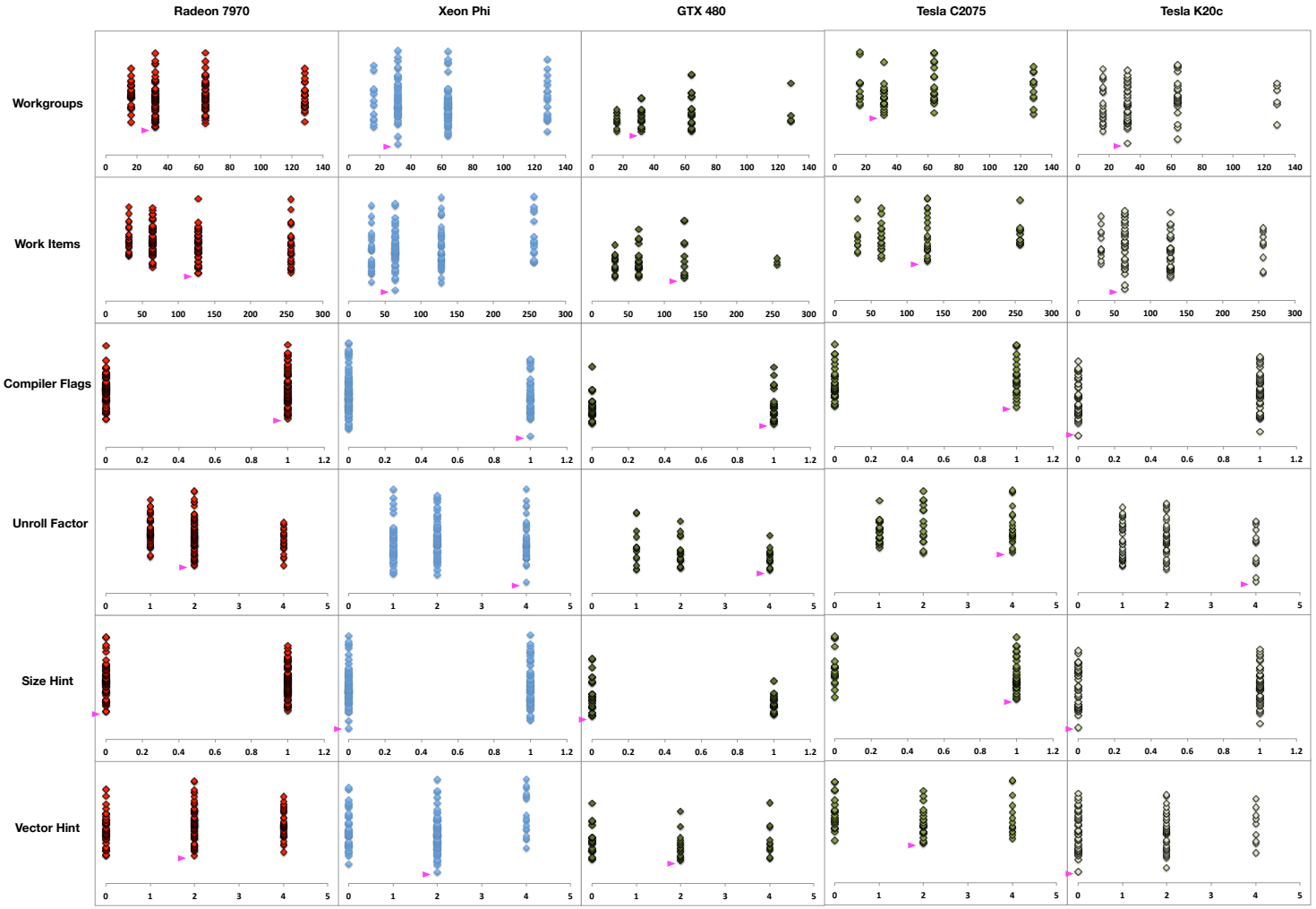

Figure 11: Performance (execution time) of the 128x128x128 FormJacobian3D kernels across architectures by tunable parameter. The point for the best version is indicated with a pink arrow.

To better understand whether certain isolated parameters account for a large proportion of improvement seen in optimized versions, we plotted all points executed during autotuning of the ex14FJ 128^3 kernel by parameter value separately for each parameter, as shown in Figure 11. Some patterns are discernible: on the Radeon 7970, for example, the best performance found improves with work items per group up to a point, and then remains flat, while on the Xeon Phi, performance improves with work items per group but then begins to degrade again. The lack of other obvious patterns indicates that performance depends in a complex manner upon the combination of the chosen transformation parameters and is not explained by any one parameter.

To better understand the causes of variation in performance across code variants, we measured hardware performance counter data where vendor libraries were provided to do so. As a small example of this, Figure 12 shows the performance of the FormFunction3D variants of all four sizes on the Intel Xeon Phi, sorted first by performance, to show the overall distribution of variants, and then sorted by `DATA_READ_MISS_OR_WRITE_MISS`. The overall trend follows the same general pattern as performance, indicating that `DATA_READ_MISS_OR_WRITE_MISS` is an important contributor to the improved performance of the faster variants.

5 Conclusions

The work here described is an inital step towards a more automated development environment for accelerator devices. To help developers write performance-portable accelerator code, we have developed a system which automatically translates annotated C code into OpenCL kernels with various optimizations applied, which we search for the best-performing variant. We showed the benefits of this approach for small numerical kernels and for radiation transport simulation code, outperforming static approaches by up to 2.5 times.

Future work will involve two approaches: in the first, we will improve the OpenCL code generator, expanding the number of supported optimizations, and

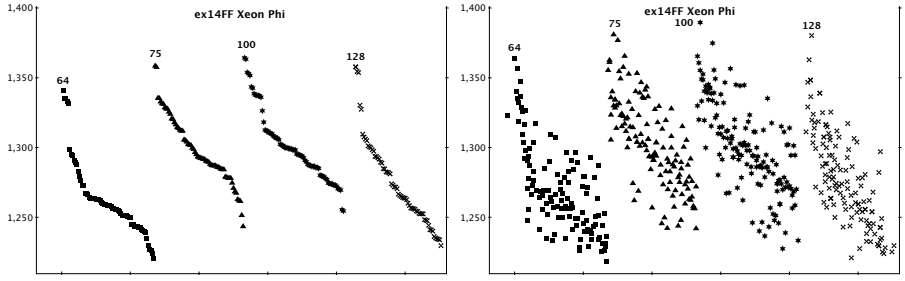

Figure 12: Execution time in milliseconds of the FormFunction3D variants for four sizes on the Xeon Phi. On left, sorted by performance; on right, same variants sorted by `DATA_READ_MISS_OR_WRITE_MISS`.

add more detailed performance analysis capabilties by integrating with instrumentation systems such as Lynx [17]. In the second, we will develop an integrated performance knowledge management system, combining performance profiles collected during autotuning with comprehensive metadata describing the code, algorithms, execution environment, and tuning methods to help developers understand the performance implications of their code.

Acknowledgement

This research is supported by the DOE X-Stack grant DE-SC0005360 and SciDAC grant DE-SC0006723. A Major Research Instrumentation grant from the National Science Foundation, Office of Cyber Infrastructure (Grant #: OCI-0960354) provided resources for our performance experiments.

References

[1] Jason Ansel et al. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. Dublin, Ireland, June 2009. URL: http://groups.csail.mit.edu/commit/papers/2009/ansel-pldi09.pdf.

[2] Prasanna Balaprakash, Stefan M. Wild, and Paul D. Hovland. "An Experimental Study of Global and Local Search Algorithms in Empirical Performance Tuning". In: *Proceedings of the 10th International Meeting on High-Performance Computing for Computational Science (VECPAR 2012)*. Kobe, Japan, July 2012.

[3] Prasanna Balaprakash, Stefan M. Wild, and Paul D. Hovland. "Can search algorithms save large-scale automatic performance tuning?" In: *Procedia Computer Science* 4 (2011). Proceedings of the International Conference on Computational Science, ICCS 2011, pp. 2136–2145. ISSN: 1877-0509. DOI: 10.1016/j.procs.2011.04.234.

[4] Prasanna Balaprakash, Stefan M. Wild, and Boyana Norris. "SPAPT: Search Problems in Automatic Performance Tuning". In: *Procedia Computer Science* 9 (2012). Proceedings of the International Conference on Computational Science, ICCS 2012, pp. 1959–1968. ISSN: 1877-0509. DOI: 10.1016/j.procs.2012.04.214.

[5] Satish Balay et al. "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries". In: *Modern Software Tools in Scientific Computing*. Ed. by E. Arge, A. M. Bruaset, and H. P. Langtangen. Birkhäuser Press, 1997, pp. 163–202.

[6] Satish Balay et al. *PETSc Web page*. http://www.mcs.anl.gov/petsc. 2013.

[7] Protonu Basu et al. "Towards making autotuning mainstream". In: *International Journal of High Performance Computing Applications* 27.4 (2013), pp. 379–393. DOI: 10.1177/1094342013493644. eprint: http://hpc.sagepub.com/content/27/4/379.full.pdf+html. URL: http://hpc.sagepub.com/content/27/4/379.abstract.

[8] R. Bell, A. Malony, and S. Shende. "A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis". In: *European Conference on Parallel Processing (EuroPar 2003)*. Vol. LNCS 2790. Sept. 2003, pp. 17–26.

[9] Nicholas Chaimov. "A Framework for Automatic Generation of Specialized Function Variants". Master's Thesis. Eugene, Oregon: University of Oregon, 2012.

[10] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. University of Utah, 2008.

[11] Jee W. Choi, Amik Singh, and Richard W. Vuduc. "Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs". In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '10. Bangalore, India: ACM, 2010, pp. 115–126. ISBN: 978-1-60558-877-3. DOI: 10.1145/1693453.1693471. URL: http://doi.acm.org/10.1145/1693453.1693471.

[12] C. Choudary et al. "Stencil-Aware GPU Optimization of Iterative Solvers". In: *SIAM Journal on Scientific Computing* (2013). To appear.

[13] Andrew Davidson and John Owens. "Toward Techniques for Auto-tuning GPU Algorithms". In: *Proceedings of the 10th International Conference on Applied Parallel and Scientific Computing - Volume 2*. PARA'10. Reykjavík, Iceland: Springer-Verlag, 2012, pp. 110–119. ISBN: 978-3-642-28144-0. DOI: 10.1007/978-3-642-28145-7_11. URL: http://dx.doi.org/10.1007/978-3-642-28145-7_11.

[14] Yuri Dotsenko et al. "Auto-tuning of fast fourier transform on graphics processors." In: *PPOPP*. Ed. by Calin Cascaval and Pen-Chung Yew. ACM, 2011, pp. 257–266. ISBN: 978-1-4503-0119-0. URL: http://dblp.uni-trier.de/db/conf/ppopp/ppopp2011.html#DotsenkoBLG11.

[15] Peng Du et al. "From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming". In: *Parallel Comput.* 38.8 (Aug. 2012), pp. 391–407. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.10.002. URL: http://dx.doi.org/10.1016/j.parco.2011.10.002.

[16] Jianbin Fang, Ana Lucia Varbanescu, and Henk J. Sips. "A Comprehensive Performance Comparison of CUDA and OpenCL." In: *ICPP*. Ed. by Guang R. Gao and Yu-Chee Tseng. IEEE, 2011, pp. 216–225. ISBN: 978-1-4577-1336-1. URL: http://dblp.uni-trier.de/db/conf/icpp/icpp2011.html#FangVS11.

[17] Naila Farooqui et al. "Lynx: A Dynamic Instrumentation System for Data-parallel Applications on GPGPU Architectures". In: *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*. ISPASS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 58–67. ISBN: 978-1-4673-1143-4. DOI: 10.1109/ISPASS.2012.6189206. URL: http://dx.doi.org/10.1109/ISPASS.2012.6189206.

[18] Matteo Frigo, Steven, and G. Johnson. "The design and implementation of FFTW3". In: *Proceedings of the IEEE*. 2005, pp. 216–231.

[19] Georgios Goumas et al. “Adapt or become extinct!: the case for a unified framework for deployment-time optimization (position paper)”. In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. EXADAPT ’11. San Jose, California: ACM, 2011, pp. 46–51. ISBN: 978-1-4503-0708-6. DOI: 10.1145/2000417.2000422. URL: http://doi.acm.org/10.1145/2000417.2000422.

[20] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. “Annotation-Based Empirical Performance Tuning Using Orio”. In: *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*. Rome, Italy, 2009.

[21] K. Huck and A. Malony. “PerfExplorer: A Performance Data Mining Framework for Large-Scale Parallel Computing”. In: *Supercomputing Conference (SC 2005)*. ACM, Nov. 2005.

[22] K. Huck et al. “Design and Implementation of a Parallel Performance Data Management Framework”. In: *International Conference on Parallel Processing (ICPP 2005)*. IEEE Computer Society, Aug. 2005.

[23] K. Huck et al. “Knowledge Support and Automation for Performance Analysis with PerfExplorer 2.0”. In: *The Journal of Scientific Programming* 16.2-3 (2008). (special issue on Large-Scale Programming Tools and Environments), pp. 123–134.

[24] J. Kurzak, S. Tomov, and J. Dongarra. “Autotuning GEMMs for Fermi”. In: (2011).

[25] Yan Li et al. “MPFFT: An Auto-Tuning FFT Library for OpenCL GPUs”. English. In: *Journal of Computer Science and Technology* 28.1 (2013), pp. 90–105. ISSN: 1000-9000. DOI: 10.1007/s11390-013-1314-8. URL: http://dx.doi.org/10.1007/s11390-013-1314-8.

[26] Kathleen A. Lindlan et al. “A tool framework for static and dynamic analysis of object-oriented software with templates”. In: *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Supercomputing ’00. Dallas, Texas, United States: IEEE Computer Society, 2000. ISBN: 0-7803-9802-5. URL: http://dl.acm.org/citation.cfm?id=370049.370456.

[27] A. Malony and et al. “Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs”. In: *International Conference on Parallel Processing (ICPP 2011)*. IEEE Computer Society, Sept. 2011, pp. 176–185.

[28] Azamat Mametjanov et al. “Autotuning Stencil-Based Computations on GPUs”. In: *Proceedings of IEEE Cluster 2012*. 2012.

[29] Gabriel Martinez, Mark Gardner, and Wu-chun Feng. “CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures”. In: *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*. ICPADS ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 300–307. ISBN: 978-0-7695-4576-9. DOI: 10.1109/ICPADS.2011.48. URL: http://dx.doi.org/10.1109/ICPADS.2011.48.

[30] P. Mucci et al. “PAPI: A Portable Interface to Hardware Performance Counters”. In: *DoD HPCMP Users Group Conference*. 1999, pp. 7–10.

[31] Rajib Nath, Stanimire Tomov, and Jack Dongarra. “Accelerating GPU Kernels for Dense Linear Algebra”. In: *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*. VECPAR’10. Berkeley, CA: Springer-Verlag, 2011, pp. 83–92. ISBN: 978-3-642-19327-9. URL: http://dl.acm.org/citation.cfm?id=1964238.1964250.

[32] S.J. Pennycook and S.A. Jarvis. “Developing Performance-Portable Molecular Dynamics Kernels in OpenCL”. In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* Nov. 2012, pp. 386–395. DOI: 10.1109/SC.Companion.2012.58.

[33] Gabe Rudy. “CUDA-CHiLL: A Programming Language Interface for GPGPU Optimization and Code Generation”. MA thesis. University of Utah, 2010.

[34] S. Shende and A. Malony. “TAU: The TAU Parallel Performance System”. In: *International Journal of High Performance Computing Applications* 20.2 (2006), pp. 287–311.

[35] Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. “Automatic Transformations for Effective Parallel Execution on Intel Many Integrated Core”. In: *TACC-Intel Highly Parallel Computing Symposium*. Austin, TX, Apr. 2012.

[36] Philippe Tillet, Karl Rupp, and Siegfried Selberherr. “An automatic OpenCL compute kernel generator for basic linear algebra operations”. In: *Proceedings of the 2012 Symposium on High Performance Computing*. HPC ’12. Orlando, Florida: Society for Computer Simulation International, 2012, 4:1–4:2. ISBN: 978-1-61839-788-1. URL: http://dl.acm.org/citation.cfm?id=2338816.2338820.

[37] Ananta Tiwari et al. “Auto-tuning full applications: A case study”. In: *Int. J. High Perform. Comput. Appl.* 25.3 (Aug. 2011), pp. 286–294. ISSN: 1094-3420. DOI: 10.1177/1094342011414744. URL: http://dx.doi.org/10.1177/1094342011414744.

[38] Antana Tiwari. “Tuning Parallel Applications in Parallel”. PhD thesis. University of Maryland, College Park, 2011.

[39] R. Clint Whaley and Jack J. Dongarra. “Automatically tuned linear algebra software”. In: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. Supercomputing ’98. San Jose, CA: IEEE Computer Society, 1998, pp. 1–27. ISBN: 0-89791-984-X. URL: http://dl.acm.org/citation.cfm?id=509058.509096.