# Toward Multi-target Autotuning for Accelerators

Nick Chaimov, Boyana Norris, and Allen Malony
*Department of Computer and Information Science*
*University of Oregon*
*Eugene, OR 97403*
{*nchaimov,norris,malony*}*@cs.uoregon.edu*

***Abstract*—Producing high-performance implementations from simple, portable computation specifications is a challenge that compilers have tried to address for several decades. More recently, a relatively stable architectural landscape has evolved into a set of increasingly diverging and rapidly changing CPU and accelerator designs, with the main common factor being dramatic increases in the levels of parallelism available. The growth of architectural heterogeneity and parallelism, combined with the very slow development cycles of traditional compilers, has motivated the development of autotuning tools that can quickly respond to changes in architectures and programming models, and enable very specialized optimizations that are not possible or likely to be provided by mainstream compilers. In this paper we describe the new OpenCL code generator and autotuner OrCL and the introduction of detailed performance measurement into the autotuning process. OrCL is implemented within the Orio autotuning framework, which enables the rapid development of experimental languages and code optimization strategies aimed at achieving good performance on new platforms without rewriting or hand-optimizing critical kernels. The combination of the new OpenCL autotuning and TAU measurement capabilities enables users to consistently evaluate autotuning effectiveness across a range of architectures, including several NVIDIA and AMD accelerators and Intel Xeon Phi processors, and to compare the OpenCL and CUDA code generation capabilities. We present results of autotuning several numerical kernels that typically dominate the execution time of iterative sparse linear system solution and key computations from a 3-D parallel simulation of solid fuel ignition.**

***Keywords*-OpenCL, TAU, autotuning, GPUs, accelerators**

## I. Introduction

To overcome the challenges posed by the limits of Dennard scaling, hardware has evolved toward increased levels of parallelism and heterogeneity, whereas prevalent programming models, languages, and the design of compilers for these languages still reflect a mostly uniform view of architectures (few homogeneous cores with a hardware-managed cache hierarchy). Given legacy languages that were not designed to express massive task and data parallelism, traditional compilers understandably lag or fail to provide portable high performance for many existing applications.

Developers wishing to exploit different hardware resources (e.g., both multicore CPUs and accelerators) have to choose between manually implementing different versions of (portions of) their applications for different platforms or underutilizing the hardware while waiting for compilers and libraries to eventually provide an optimized solution. Moreover, manual architecture specialization is costly in terms of human effort and decreases readability and reliability.

Annotation-based approaches such as OpenMP [19] and OpenACC [1] attempt to compensate for some of the deficiencies in existing programming languages, by enabling developers to explicitly designate parallelizable portions of code, but because their scope is limited, the resulting performance is also limited. Frequently annotation-based approaches (which aim to reduce the changes required to the original code) require a significant restructuring of the implementation in order to achieve better performance. Similar to manual tuning, such changes can reduce code readability, maintainability, and performance portability to a different architecture.

To address the disparity between the features architectures are providing and what compilers and other tools can effectively exploit, we have been developing the Orio framework, which enables rapid implementation of code generators for multiple architectural targets through source-to-source transformations and empirical performance optimization (prior to this work, supported targets included C, Fortran, and CUDA).

This paper presents the following new contributions.

- We have enabled detailed low-overhead performance measurements during the autotuning process with the TAU performance analysis framework [14], which support more detailed analysis of the effect of transformations on different performance metrics.
- We have implemented a new OpenCL [21] autotuning code generator, OrCL, which can be used to produce parallel code for GPUs, Intel MIC processors, and multicore CPUs.
- We present a cross-language (OpenCL and CUDA) and cross-architecture comparison of the results for autotuning several kernels responsible for significant portions of the execution time of scientific applications.

The rest of the paper is organized as follows. In Section II we briefly discuss relevant prior work. In Section III, we describe the new OrCL tool implementation and the integration of fine-grained performance measurements into the autotuning framework. Section IV presents results from evaluating OrCL on several architectures through autotuning linear algebra kernels from Newton-Krylov solvers and the stencil-based updates and Jacobian computations of a solid fuel ignition application. Section V presents our conclusions and outlines future work.

## II. Background

OrCL was implemented within the Orio open-source, extensible framework for the definition of domain-specific languages and generation of optimized code for multiple architecture targets, including support for empirical autotuning of the generated code. The workflow of the Orio framework is illustrated in Figure 1. In previous work, we demonstrated that relatively high-level computation specifications can be embedded in existing C or Fortran codes through annotations expressed as structured comments [17]. Based on these simple (typically loop-based) computation specifications and an additional tuning specification that contains a list of possible transformations and their parameters, Orio generates optimized versions of the computation in C, Fortran, or CUDA. The performance of different versions is evaluated empirically; however, because the search space of all possible transformations and their parameters is generally too large to test exhaustively, Orio supports a number of search strategies which dramatically reduce the number of variants that must be compiled and run. In addition to exhaustive and random search, several search algorithms are available, including two variants of Nelder-Mead simplex search (adapted to discrete problems), a chaos genetic algorithm, and simulated annealing [2]. The performance of the generated and autotuned code typically exceeds that of compiled C or Fortran code, and by exploiting temporal locality for composed operations, autotuned implementations frequently outperform sequences of calls to optimized numerical libraries such as vendor versions of the BLAS and LAPACK [7], [8], [18].

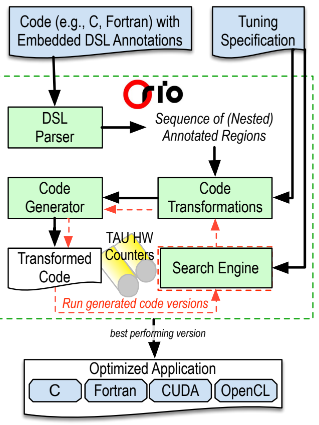

Figure 1. Orio workflow.

Similar to most autotuners we are aware of, Orio measured only execution time (the objective function of the optimization process), which is the coarsest possible measurement of the effects of different optimizations and is typically not sufficient to reason about the impact of a particular code transformation or parameter value. To enable the more fine-grained measurements required to inform the autotuner about the specific effects of optimizations (e.g., reduction in L1 cache misses for a tiling transformation), we integrated Orio with the TAU Performance System® [20] and its portable instrumentation, measurement, and analysis environment [14] for parallel applications and machines. TAU operates with both the CUDA and OpenCL accelerator programming languages and can obtain performance measurements in hybrid parallel applications on scalable heterogeneous platforms [13].

In addition to providing portable interfaces to instrumentation and measurement capabilities, the TAU performance analysis environment includes a performance data management system, *TAUdb* [10], that stores parallel profiles from performance experiments along with their associated metadata. TAUdb provides an API for storing and retrieving information from the database, which all TAU analysis tools use. The *ParaProf* [5] parallel profile analyzer enables detailed study and comparison of performance profiles for individual performance experiments, while the *PerfExplorer* [9], [11] performance data mining framework in TAU can apply performance analysis operations across multiple experiments.

Generation of OpenCL implementations from high-level specifications is also provided by the The Vienna Computing Library (ViennaCL) [22], which is an open-source scientific computing library written in C++ that provides CUDA, OpenCL and OpenMP computing backends. While ViennaCL relies on an operator-overloading C++ expression template approach, OrCL employs source-to-source transformation. Some compilers target accelerator code generation guided by pragma-based annotations such as OpenACC, which enable programmers to indicate parallelism within existing sequential implementations. By contrast, OrCL requires users to rewrite the computations to be tuned (i.e., extracting and simplifying a loop-based implementation), effectively removing unrelated context and control flow information and thus enabling more aggressive optimization.

## III. Cross-Architecture Autotuning with OpenCL

By implementing OpenCL code generation and autotuning support in Orio, we have enabled the creation of high-performance implementations for *different hardware targets*. While OpenCL provides a standard and portable language, a single manual implementation will exhibit different performance on different architectures for which OpenCL compilation and runtime environments are available. Notably, the mapping of OpenCL memory spaces onto physical device memories vary between types of devices, vendors, and generations of devices within a vendor. OpenCL, by abstracting the memory hierarchy, makes it somewhat difficult to infer from the code what data will be placed in a register, in cache, or in main memory. Hence, exploring different optimizations through autotuning is essential for obtaining consistent high performance across heterogeneous architectures.

### A. The OpenCL Programming Model

The goal of OpenCL is to enable developers to write a portable program once and deploy it on any heterogeneous system. The OpenCL standard [21] defines a programming environment and model for specifying parallel general-purpose computations for execution on multicore processors and accelerator devices. Unlike NVIDIA's CUDA [16], which is designed for use only with NVIDIA's GPUs, OpenCL is not designed with any one vendor, device, or architecture in mind. Each vendor wishing to provide OpenCL support on its devices provides its own implementation of the OpenCL standard, mapping the OpenCL programming model onto the particular architecture of the targeted devices.

In the OpenCL programming model, a program is divided into two components: *host* and *device*. The host is responsible for initializing the OpenCL environment, allocating device memory and copying data between the host and device, invoking kernel functions which execute on the device, and synchronizing between kernel invocations. These are accomplished by means of *command queues*, into which copy, kernel invocation, and synchronization commands are placed.

On the device, kernel executions are partitioned into *workgroups*, which in turn are composed of *work items*. Each work item in a workgroup executes the same code, and each workgroup executes independently of any other workgroups which may be executing on the same device. Work items have access to *private* memory, while all the work items in a workgroup share *local* memory. The set of all concurrently-executing workgroups then share read-write *global* and read-only *constant* memory. The host is responsible for specifying how the overall work to be performed is to be partitioned by specifying the number of workgroups and the number of work items per workgroup, which may be limited by architectural features of the particular device being used.

### B. OpenCL Code Generation with Orio

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 designed and implemented the OrCL code generator 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 Orio module described in [15] 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 from the same high-level specification of the computation.

```
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 2. Annotated vecAXPY kernel for OpenCL generation.

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 tun-

```
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 3. Example of a generated OpenCL vecAXPY kernel.

ing specification (see Figure 2) 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 OrCL is invoked on the code, variants are generated (e.g, see Figure 3) and tested based on the specification. 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. The parameter values used by OrCL are described below.

*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 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 required size, allowing the compiler to make optimizations that would yield incorrect results for other sizes. OpenCL compilers, 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 splits the execution into multiple, overlapping transfers and kernel executions by using CUDA asynchronous streams. This

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

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. We also do not currently support changes to data layout, such as converting between structure-of-arrays and array-of-structures formats, or between row-major and column-major arrays.

## C. Performance Measurement with TAU

For each requested combination of values of the above parameters, OrCL generates OpenCL device and host code. Figure 4 shows the generated host code for invoking the OpenCL vecAXPY kernel. 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 [20], 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, when available. 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 [13]. For CUDA, hardware performance counters can be sampled using CUPTI. For OpenCL, timings from OpenCL events are captured. Hardware performance counters are captured for AMD cards via AMD's GPUPerfAPI and for the Xeon Phi via Intel's VTune. At this time, NVIDIA does not provide any mechanism for capturing hardware performance counters when running OpenCL code.

For each variant tested, performance information gathered by TAU is stored into TAUdb, a database system for storing performance profiles and related metadata [10]. 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). Figure 5 shows the data flow between Orio and TAU.

Data stored in TAUdb can be analyzed using the *ParaProf* visualization system [5] and the *PerfExplorer* data mining framework [9], [11]. PerfExplorer analyses can be performed with a GUI or scripted by using a Python interface. By using this new automated fine-grained data collection capability and preliminary work on using machine learning-based performance models to select the starting point for the search [6], we plan to automate the generation of these models and use them not just for search initialization, but also to reduce the overall search space of possible transformations during the autotuning process.

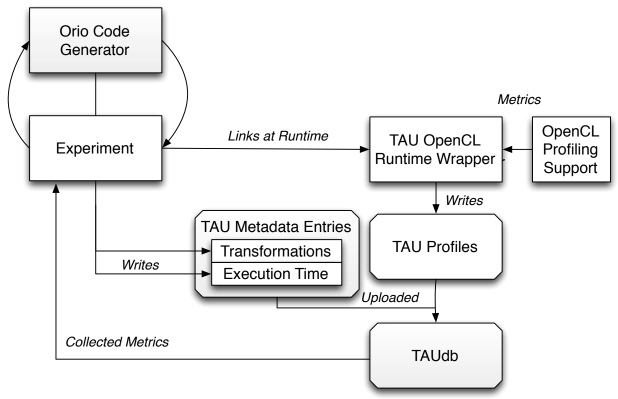

Figure 5. Data flow between Orio, TAU, and TAUdb.

## IV. Evaluation

The computations we autotuned were selected among the functions that dominate the execution time of applications based on the solution of nonlinear partial differential algorithms (PDEs) discretized on a regular grid and solved by using Newton-Krylov iterative methods. For example, the implementations of most Newton-Krylov nonlinear solvers consist of sparse matrix-vector products and variations of the AXPY operation. In Jacobian-free Newton-Krylov approaches [12], the Jacobian-vector product approximation required for preconditioning is computed through repeated evaluation of the application-specific stencil-update function or user-provided Jacobian (in PETSc [4], for example, these are the `FormFunction` and `FormJacobian` functions – the latter typically dominates the execution time for most PDE-based applications). Table I lists the linear algebra kernels we considered. In addition, we also evaluated automatically generated and tuned version of the function and Jacobian computations for a real application simulating 3-D solid fuel ignition (Figures 8–11).

For the linear algebra computations in Table I, 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

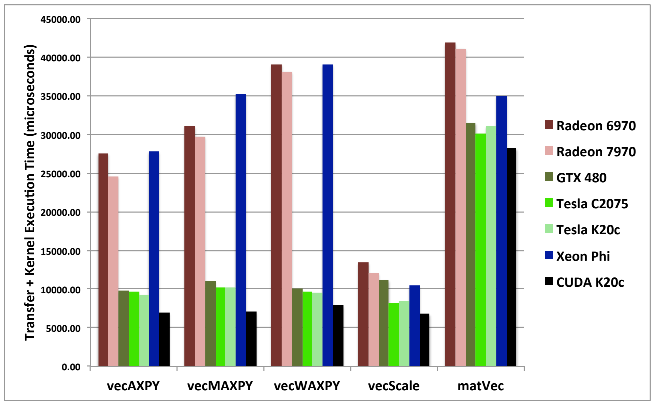

Figure 6. Absolute performance of the best-performing linear algebra kernel variant across devices. The first six bars in each group correspond to code produced with OrCL, while the last bar is code produced with OrCUDA.

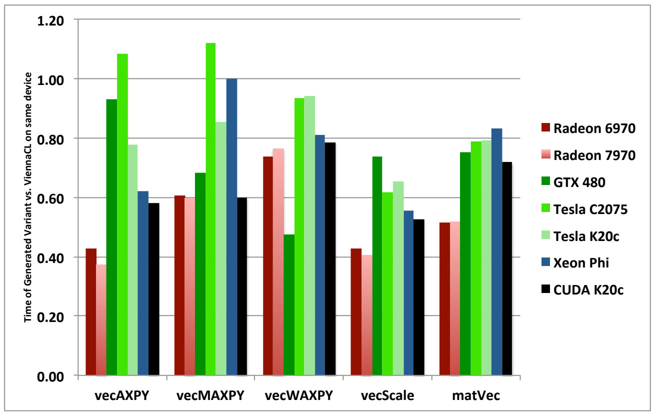

Figure 7. Performance of the best-performing linear algebra kernel variant normalized by time to run the corresponding ViennaCL version on the same device. The first six bars in each group correspond to code produced with OrCL, while the last bar is code produced with OrCUDA.

scalars.

Table I
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 | $w = y + \alpha x$ |

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 given in Table IV. 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 III-C.

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 the best CUDA variant found for the K20c GPU. Figure 7 shows the performance of the same variants, normalized by the execution time for the equivalent ViennaCL [22] 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 [3], [4] application solving a 3-D solid fuel ignition (SFI) problem, defined as the following boundary value problem

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

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 invoke the application-specific function and Jacobian computations [1], which we designate as EX14FF and EX14FJ, respectively. The OrCL input for the main loop in these functions is shown in Figure 8. The performance of the PETSc iterative solvers also heavily depends on the linear algebra kernels in Table I (and a few others). Most stencil-based computations can be similarly optimized by OrCL, which is not limited to matrix algebra. We autotuned the function and Jacobian computations 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, single variants are more expensive to evaluate and the search space for each input size consists of 7 parameters resulting in a total number of 5,760 variants of varying individual costs for EX14FF and EX14FJ. These more complex functions produced much more varied autotuning results.

Figure 9 shows the sorted execution times for all OpenCL code variants generated by the search for each of the kernels and devices for the two largest sizes. The Radeon 6970 is omitted from results for all but the ex14FF $64^3$ because that card did not 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. To better understand the causes of variation in performance across code variants, we measured hardware performance

```
/*@ 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]);
    }
  }

) @*/
```

```
/*@ 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 and ex14FF computations.

[1]These are user-provided implementations that can be of arbitrary size and complexity and are not typically considered *kernels*.

Table II
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 WG Size Multiple | 64 | 64 | 32 | 32 | 32 | 16 |

Table III
PARAMETER VALUES FOR THE BEST PERFORMING EX14FF AND EX14FJ FUNCTIONS 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) |

counter data where vendor libraries were provided to do so. As a small example of this, Figure 11 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.

## V. CONCLUSION

It is a widely held belief that the ability to program current- and next-generation application accelerators portably and optimally is now beyond the expertise of most humans. However, the ability to write algorithms in a simple language syntax that can be transformed to multiple target languages and autotuned for different accelerator is is not. Our research contributes the integration of a code transformation and autotuning framework with a robust empirical performance analysis toolkit for the goal of advancing the state of the art in automated accelerator code generation, characterization, and optimization. The addition of OpenCL to Orio's output languages significantly expands the possible accelerator platforms it can target. By incorporating TAU's portable, multi-architecture measurement techniques, performance data management system, and programmable analysis tools, empirical characterization and evaluation is available to inform and improve the autotuning process.

We have described the new OrCL code generator and autotuner for OpenCL, which obtains very good results on a variety of accelerator platforms and outperforms static approaches by up to a factor of 2.5. We plan to continue expanding Orio's code generation capabilities, for example,

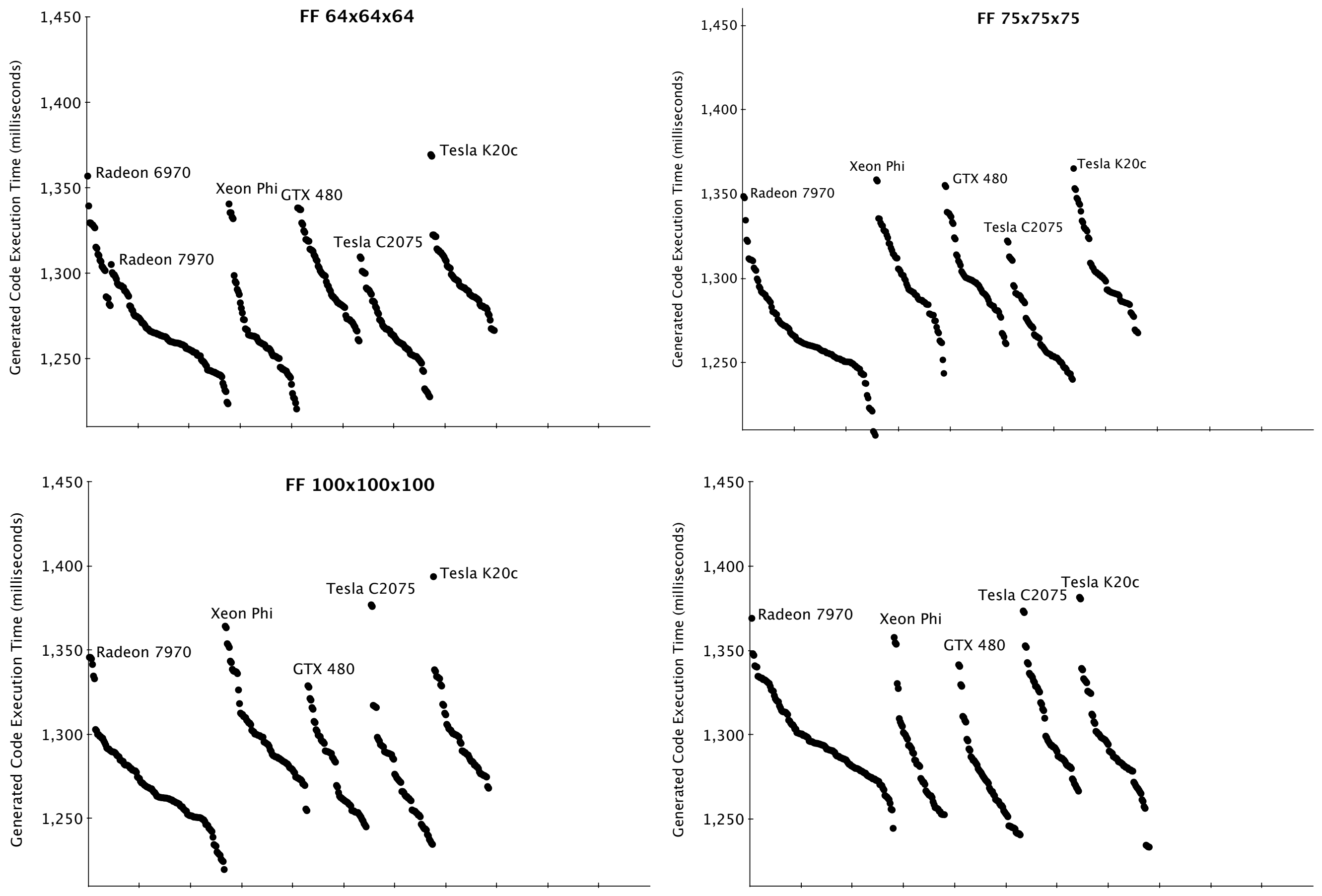

Figure 9. Performance (execution time in milliseconds) of evaluated variants for FormFunction3D kernels of the two largest sizes. The X-axis represents different variants executed during the autotuning for each case studied.

by adding generation for OpenACC directives to our current CPU backends (C and Fortran) and providing this as one of the non-CPU autotuning targets. Because Orio also enables the definition of simple domain-specific input languages, we plan to explore additional optimizations that can be performed at a higher semantic level and would not be possible for lower level-inputs (e.g., C). For example, if the domain is stencil-based grid computations, Orio can exploit the known sparsity structure of the sparse matrix used in the solution of nonlinear PDEs discretized on a regular grid to produce optimized matrix and vector kernels that have regular memory access patterns without indirection and are thus much more amenable to optimizations than general-purpose sparse matrix representations (e.g., those using compressed sparse row data structures).

There are several other directions our work is poised to pursue. The range of supported OpenCL code transformations will be expanded as we gain more experience with the language. There are opportunities to expand the scope of accelerator architecture and compiler parameters for autotuning. We will tackle larger autotuning challenges, such as generating autotuned versions for the complete BLAS, or applying it to more complex computations. This will also allow more rigorous baseline comparisons with existing optimized products. Last, we will add more sophisticated data mining capabilities to the environment for factor analysis, feature extraction, and correlation computations. Additionally, we will investigate machine learning to capture knowledge about optimization strategies and their relationships to code and architectural properties.

## Acknowledgments

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] OpenACC: Directives for accelerators. http://www.openacc-standard.org. Last accessed January 11, 2013.

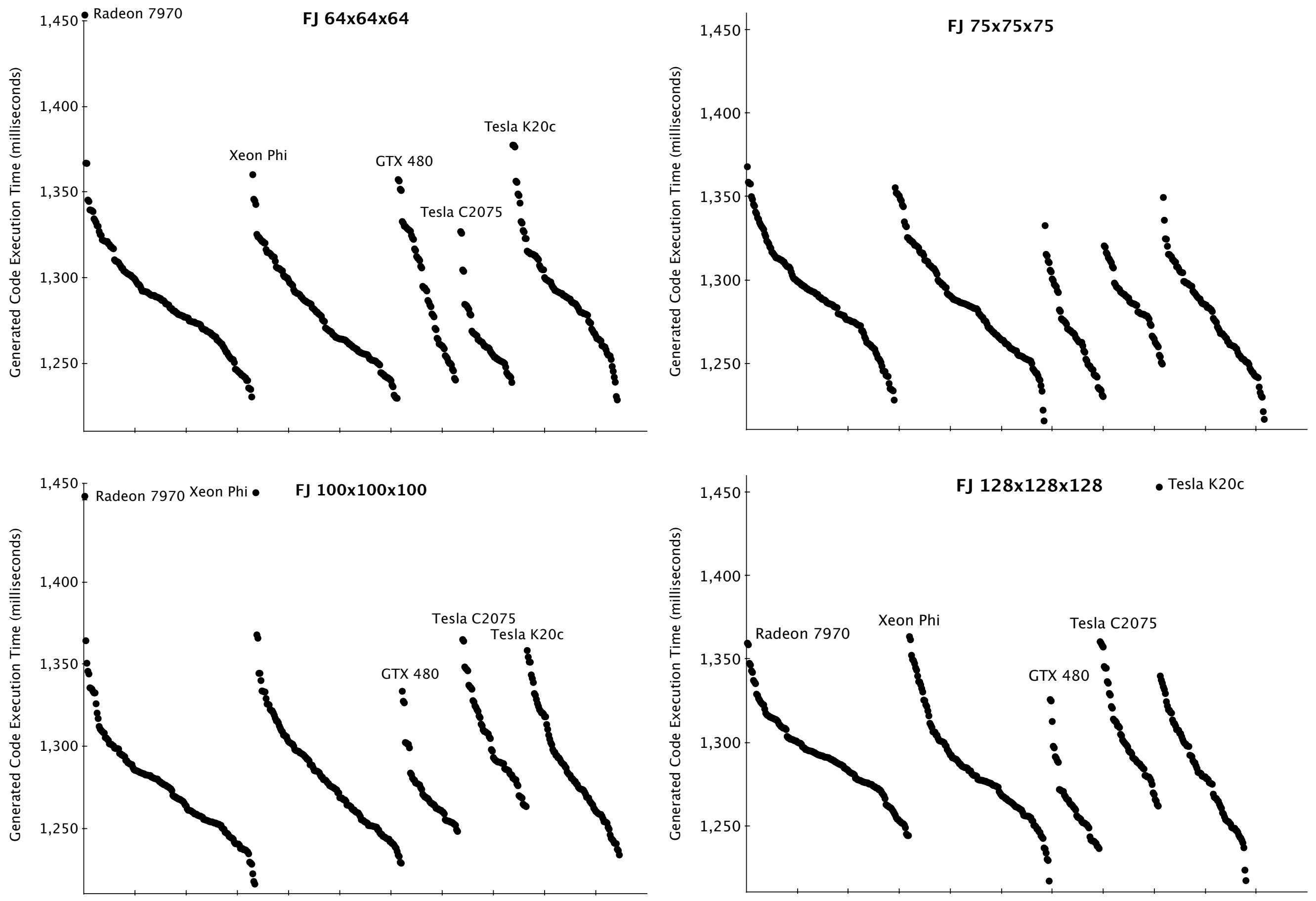

Figure 10. Performance (execution time in milliseconds) of evaluated variants for FormJacobian3D kernels of four sizes. The X-axis represents different variants executed during the autotuning for each case studied.

[2] BALAPRAKASH, P., WILD, S. M., AND HOVLAND, P. D. An experimental study of global and local search algorithms in empirical performance tuning. In *High Performance Computing for Computational Science - VECPAR 2012, LNCS* (2013), vol. 7851, p. 261269.

[3] BALAY, S., BROWN, J., BUSCHELMAN, K., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., SMITH, B. F., AND ZHANG, H. PETSc Web page, 2013. http://www.mcs.anl.gov/petsc.

[4] BALAY, S., GROPP, W. D., MCINNES, L. C., AND SMITH, B. F. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing* (1997), E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhäuser Press, pp. 163–202.

[5] BELL, R., MALONY, A., AND SHENDE, S. A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *European Conference on Parallel Processing (EuroPar 2003)* (Sept. 2003), vol. LNCS 2790, pp. 17–26.

[6] CHAIMOV, N., BIERSDORFF, S., AND MALONY, A. D. Tools for machine-learning-based empirical autotuning and specialization. *International Journal of High Performance Computing Applications 27*, 4 (2013), 403–411.

[7] CHOUDARY, C., GODWIN, J., HOLEWINSKI, J., KARTHIK, D., LOWELL, D., MAMETJANOV, A., NORRIS, B., SABIN, G., AND SADAYAPPAN, P. Stencil-aware GPU optimization of iterative solvers. *SIAM Journal on Scientific Computing* (2013). To appear.

[8] HARTONO, A., BASKARAN, M. M., BASTOUL, C., COHEN, A., KRISHNAMOORTH, S., NORRIS, B., RAMANUJAM, J., AND SADAYAPPAN, P. PrimeTile: A parametric multi-level tiler for imperfect loop nests. In *Proceedings of the 23rd International Conference on Supercomputing, June 8-12, 2009, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA* (2009). Also available as Tech. Report OSU-CISRC-2/09–TR04.

[9] HUCK, K., AND MALONY, A. PerfExplorer: A Performance Data Mining Framework for Large-Scale Parallel Computing. In *Supercomputing Conference (SC 2005)* (Nov. 2005), ACM.

[10] HUCK, K., MALONY, A., BELL, R., AND MORRIS, A. Design and Implementation of a Parallel Performance Data Management Framework. In *International Conference on*

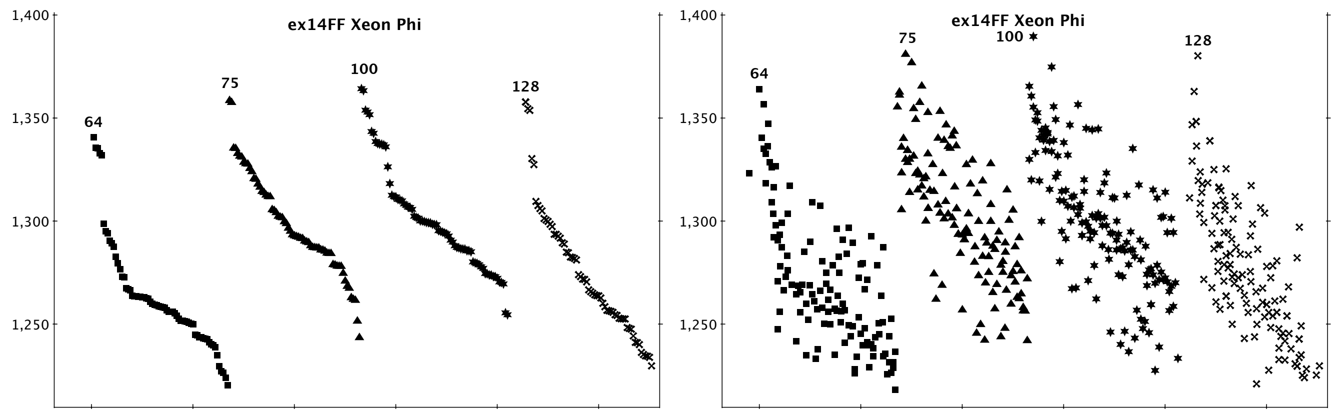

Figure 11. Execution time in milliseconds of the FormFunction3D variants for four sizes on the Xeon Phi. On left, sorted by performance; on right, same variants sorted by `DATA_READ_MISS_OR_WRITE_MISS`. The X-axis represents different variants executed during the autotuning for each case studied.

*Parallel Processing (ICPP 2005)* (Aug. 2005), IEEE Computer Society.

[11] Huck, K., Malony, A., Shende, S., and Morris, A. Knowledge Support and Automation for Performance Analysis with PerfExplorer 2.0. *The Journal of Scientific Programming 16*, 2-3 (2008), 123–134. (special issue on Large-Scale Programming Tools and Environments).

[12] Knoll, D. A., and Keyes, D. E. Jacobian-free Newton-Krylov methods: A survey of approaches and applications. *J. Comput. Phys. 193*, 2 (Jan. 2004), 357–397.

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

[14] Malony, A., Mellor-Crummey, J., and Shende, S. Methods and Strategies for Parallel Performance Measurement and Analysis: Experiences with TAU and HPCToolkit. In *Performance Tuning of Scientific Applications*, D. Bailey, R. Lucas, and S. Williams, Eds. CRC Press, New York, 2010.

[15] Mametjanov, A., Lowell, D., Ma, C.-C., and Norris, B. Autotuning stencil-based computations on GPUs. In *Proceedings of IEEE Cluster 2012* (2012).

[16] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with CUDA. *Queue 6*, 2 (Mar. 2008), 40–53.

[17] Norris, B., Hartono, A., and Gropp, W. Annotations for productivity and performance portability. In *Petascale Computing: Algorithms and Applications*, Computational Science. Chapman & Hall / CRC Press, Taylor and Francis Group, 2007, pp. 443–462. Preprint ANL/MCS-P1392-0107.

[18] Norris, B., Hartono, A., Jessup, E., and Siek, J. Generating empirically optimized composed matrix kernels from MATLAB prototypes. In *Proceedings of the International Conference on Computational Science 2009* (2009).

[19] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[20] Shende, S., and Malony, A. TAU: The TAU Parallel Performance System. *International Journal of High Performance Computing Applications 20*, 2 (2006), 287–311.

[21] Stone, J. E., Gohara, D., and Shi, G. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test 12*, 3 (May 2010), 66–73.

[22] Tillet, P., Rupp, K., and Selberherr, S. An automatic OpenCL compute kernel generator for basic linear algebra operations. In *Proceedings of the 2012 Symposium on High Performance Computing* (San Diego, CA, USA, 2012), HPC '12, Society for Computer Simulation International, pp. 4:1–4:2.