Autotuning Stencil-Based Computations on GPUs

Azamat Mametjanov*, Daniel Lowell*, Ching-Chen Ma†, Boyana Norris*
* Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439
{azamat,dlowell,norris}@mcs.anl.gov
† Computer Science & Software Engineering
Rose-Hulman Institute of Technology, Terre Haute, IN 47803
mac@rose-hulman.edu

***Abstract*—Finite-difference, stencil-based discretization approaches are widely used in the solution of partial differential equations describing physical phenomena. Newton-Krylov iterative methods commonly used in stencil-based solutions generate matrices that exhibit diagonal sparsity patterns.To exploit these structures on modern GPUs, we extend the standard diagonal sparse matrix representation and define new matrix and vector data types in the PETSc parallel numerical toolkit. We create tunable CUDA implementations of the operations associated with these types after identifying a number of GPU-specific optimizations and tuning parameters for these operations. We discuss our implementation of GPU autotuning capabilities in the Orio framework and present performance results for several kernels, comparing them with vendor-tuned library implementations.**

***Index Terms*—autotuning, stencil, CUDA, GPU**

I. Introduction

Exploiting hybrid CPU/GPU architectures effectively typically requires reimplementation of existing CPU codes. Furthermore, the rapid evolution in accelerator capabilities means that GPU implementations must be revised frequently to attain good performance. One approach to avoiding such code reimplementation and manual tuning is to automate CUDA code generation and tuning. In this paper, we introduce a preliminary implementation of a CUDA backend in our Orio autotuning framework, which accepts a high-level specification of the computation as input and then generates multiple code versions that are empirically evaluated to select the best-performing version for given problem inputs and target hardware. In our prototype, we target key kernels in the PETSc parallel numerical toolkit, which is widely used to solve problems modeled by nonlinear partial differential equations (PDEs).

A. Motivation

Increasing heterogeneity in computer architectures at all scales presents significant new challenges to effective software development in scientific computing.

Key numerical kernels in high-performance scientific libraries such as Hypre [11], PETSc [3], [4], [5], SuperLU [13], and Trilinos [22] are responsible for much of the execution time of scientific applications. Typically, these kernels implement the steps of an iterative linear solution method, which is used to solve the linearized problem by using a family of Newton-Krylov methods. In order to achieve good performance, these kernels must be optimized for each particular architecture. Automating the generation of highly optimized versions of key kernels will improve both application performance and the library developers' productivity. Furthermore, libraries can be "customized" for specific applications by generating versions that are optimized for specific input and use characteristics.

Traditionally, numerical libraries are built infrequently on a given machine, and then applications are linked against these prebuilt versions to create an executable. While this model has worked well for decades, allowing the encapsulation of sophisticated numerical approaches in application-independent, reusable software units, it suffers from several drawbacks to achieving high performance on modern architectures. First, it provides a partial view of the implementation (either when compiling the library or the application code using it), limiting potential compiler optimizations. Because the library is built without any information on how exactly it will be used, many potentially beneficial optimizations are not considered. Second, large toolkits such as PETSc and Trilinos provide many configuration options whose values can significantly affect application performance. Blindly using a prebuilt library can result in much lower performance than achievable on a particular hardware platform. Even when a multitude of highly optimized methods exist, it is not always clear which implementation is most appropriate in a given application context. For example, the performance of different sparse linear solvers varies for linear systems with different characteristics.

Our goal is to tackle the challenges in achieving the best possible performance in the low-level fundamental kernels that many higher-level numerical algorithms share through *application-aware* code generation and tuning. Several components are required in order to provide these code generation capabilities.

- A mechanism for defining the computation at a sufficiently high level
- Application-specific library optimizations
- Automatic code generation and tuning of computationally significant library and application functionality
- Ability to use nongenerated (manual) implementations when desired

B. Background

This work relies on and extends two software packages: the autotuning framework Orio and the Portable, Extensible Tolkit for Scientific Computation (PETSc). We describe them briefly in this section.

1) Orio: Orio is an extensible framework for the definition of domain-specific languages, including support for empirical autotuning of the generated code. In previous work we have shown that high-level computation specifications can be embedded in existing C or Fortran codes by expressing them through annotations specified as structured comments [10], [17], as illustrated in Figure 1. The performance of code generated from such high-level specifications is almost always significantly better than that of compiled C or Fortran code and for composed operations it far exceeds that of multiple calls to optimized numerical libraries.

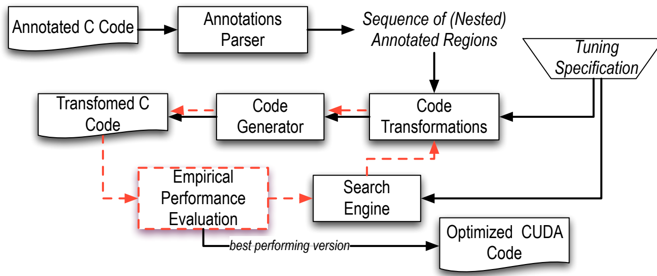

Fig. 1. Orio autotuning process.

2) PETSc: PETSc [3], [4], [5] is an object-oriented toolkit for the numerical solution of nonlinear PDEs. Solvers, as well as data types such as matrices and vectors, are implemented as objects by using C. PETSc provides multiple implementations of key abstractions, including vector, matrix, mesh, and linear and nonlinear solvers. This design allows seamless integration of new data structures and algorithms into PETSc while reusing most of the existing parallel infrastructure and implementation without requiring modification to application codes. In terms of applications, our focus is on finite-difference, stencil-based approximations, supported by the Newton-Krylov solvers in PETSc, which solve nonlinear equations of the form $f(u) = 0$, where $f : R^n \to R^n$, at each timestep (for time-dependent problems). The time for solving the linearized Newton systems is typically a significant fraction of the overall execution time. This motivates us to consider the numerical operations within the Krylov method implementations as the first set of candidates for code generation and tuning.

C. Approach

In this paper we present our approach to addressing several of the challenges introduced in Section I-A. Existing implementations of key kernels (e.g., matrix-vector products) can be generated from relatively simple specifications, either in a domain language such as MATLAB or by expressing them as simple C loops operating over arrays. Previous work on generating optimized code for composed linear algebra operations [17] demonstrates that the generated code performance can greatly exceed that of collections of calls to prebuilt libraries. The key challenges are to associate different underlying matrix and vector representations with the high-level kernel specification. At a high level, our solution consists of the following steps.

- Define optimized data structures for stencil-based computations.
- Create high-level specifications for key numerical kernels.
- Implement CUDA-specific transformations and code generation.

From the point of view of a library or application developer, using these capabilities requires the following steps.

- Provide application-relevant inputs to numerical kernels.
- Specify desired transformations and associated parameters.
- Perform empirical tuning on the target architecture.
- Build the library and application incorporating tuned kernels and use for production runs.

The first two steps can be at least partly automated, as discussed briefly in Section VI; but at present, the input specification is manual. In this paper we focus on the implementation of the CUDA code generation and autotuning capabilities.

The main goal of this work is to enable sustainable high performance on a variety of architectures while reducing the development time required for creating and maintaining library and application codes without requiring complete rewriting of substantial portions of existing implementations. While for a small set of numerical kernels a vendor-tuned library provides a great solution, available tuned libraries do not cover the full space of functionality required by applications. Hence, we are pursuing code generation and autotuning as a *complementary* solution, which can be used when vendor-supplied libraries do not satisfy the performance or portability needs of an application.

a) Summary of contributions: In this paper, we present new functionality implemented in the Orio framework that supports the following.

- Automation of code optimizations targeting GPUs that exploit the structure present in stencil-based computations
- GPU autotuning approach that can be integrated into traditional C/C++ library development
- Ability to generate specialized library versions tuned for specific application requirements

II. Stencil-Based Data Structures

This work was motivated by initial exploration of compact matrix representations for stencil-based methods [9]. Here, we briefly summarize the standard data structures used in stencil-based methods.

Finite-difference methods approximate the solution of a differential equation by discretizing the problem domain and approximating the solution by computing the differences of the model function values at neighboring inputs based on one of several possible *stencils*. An example is the heat equation where the domain is uniformly partitioned and the temperature

is approximated at discrete points. Adding a time dimension to the problem domain provides a model of heat dissipation. The standard approach to solving such problems is to apply stencils at each point such that the temperature at a point in one timestep depends on the temperature of a set of neighboring points in a previous time step. The set of neighboring points is determined by the dimensionality of the problem domain $d \in \{1,2,3\}$, the shape of a stencil $s \in \{star, box\}$, and the stencil's width $w \in N$.

For example, in a star-shaped stencil of width one applied to a two-dimensional domain (2dStar1), each point interacts with four of its neighbors to the left, right, above, and below its position within the grid. Depending on the domain's model, the points on the boundaries of the grid can satisfy Dirichlet, periodic, or other boundary conditions.

Stencil-based point interactions are captured in an adjacency (Jacobian) matrix, which is very sparse for small-size stencils. The sparsity pattern is diagonal, with the main diagonal capturing self-interactions of grid points. Interactions with other grid points are represented by diagonals that are offset to the left or right of the main diagonal. For example, the 2dStar1 stencil will generate a matrix with five diagonals—two at each side of the main diagonal.

In order to store and use the Jacobian matrix efficiently in iterative updates, the matrix can be compressed into the diagonal (DIA) format represented by two arrays: a two-dimensional *values* array, where each column represents a diagonal, and a one-dimensional *offsets* array, which stores offsets of diagonals from the main diagonal. Negative offset values represent subdiagonals, and positive offsets represent superdiagonals: for example, $[-1, 0, 1]$ for the 1dStar1 stencil. Off-diagonals are padded at the top or bottom to ensure uniform height.

The primary advantage of this storage format is the reduction of the memory footprint of the matrix due to the implicit column indexing. The column index of an element within a diagonal is computed by adding to the element's (row) index the offset of the diagonal. For example, the element at position 4 in the leftmost diagonal of a 1dStar1 stencil has matrix index $(4, 3)$.

Figure 2 illustrates a two-dimensional grid G, its corresponding 2dStar1 adjacency matrix A, and the matrix representation in DIA format. Entries marked with @ represent placeholders for neighbors under Dirichlet or periodic boundary conditions.

Note that the standard DIA format for Dirichlet boundary conditions explicitly stores values used to pad the diagonals for uniform height. In this work, we reduce DIA's memory footprint further and do not store the padding values that lie outside the matrix.

The savings in storage can be substantial. For example, given a grid with dimensions $dims = [m, n, p]$, the height of the main diagonal of a matrix is $m * n * p$. In the typical case of a star-shaped width-1 stencil with degrees of freedom dof, the amount of padding for a D-dimensional grid is the following.

$$G = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 2 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 2 & 3 & @ & 0 & 6 & 0 & 0 & 0 \\ 1 & 0 & @ & 4 & 5 & 0 & 7 & 0 & 0 \\ 0 & 2 & 0 & 4 & 5 & 6 & 0 & 8 & 0 \\ 0 & 0 & 3 & 0 & 5 & 6 & @ & 0 & 9 \\ 0 & 0 & 0 & 4 & 0 & @ & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 & 7 & 8 & 9 \\ 0 & 0 & 0 & 0 & 0 & 6 & 0 & 8 & 9 \end{bmatrix}$$

$$values = \begin{bmatrix} \underline{@} & \underline{@} & 1 & 2 & 4 \\ \underline{@} & 1 & 2 & 3 & 5 \\ \underline{@} & 2 & 3 & @ & 6 \\ 1 & @ & 4 & 5 & 7 \\ 2 & 4 & 5 & 6 & 8 \\ 3 & 5 & 6 & @ & 9 \\ 4 & @ & 7 & 8 & \underline{@} \\ 5 & 7 & 8 & 9 & \underline{@} \\ 6 & 8 & 9 & \underline{@} & \underline{@} \end{bmatrix}$$

$$offsets = [-3, -1, 0, 1, 3]$$

Fig. 2. Compressed DIA format.

D	Size of *values*	Padding
1	m*dof*3	P_1=2*dof
2	m*n*dof*5	P_2=P_1+2*m*dof
3	m*n*p*dof*7	P_3=P_2+2*m*n*dof

A. *SeqDia: A New PETSc Matrix Implementation*

Having extended the standard DIA sparse matrix representation, we implemented a new PETSc matrix data type. This implementation is similar to PETSc's AIJ sparse matrix format in that it defines the same abstract matrix data type and operations; however, the compression scheme is based on the DIA, rather than CSR, matrix storage format. Standard matrix operations such as matrix-vector multiplication $y = Ax$ (MatMult) can take advantage of the diagonal sparsity structure of the matrix A using the new implementation. An application that uses PETSc can choose the new matrix implementation by selecting the command-line option *-mat_type seqdia* at runtime (no application implementation change is required).

III. Autotuning on GPUs

One of the leading frameworks for programming GPUs is the CUDA parallel programming model [16]. In this model, an application consists of a *host* function that allocates resources on a GPU *device*, transfers data to the device, and invokes *kernel* functions that operate on the data in the single-instruction, multiple-data (SIMD) manner. At invocation, the kernel is programmed to be executed by a *grid* of *thread blocks*. The CUDA runtime manages the details of scheduling *warps* (groups of 32 threads) to execute on the device.

A typical NVIDIA device is organized as an array of streaming multiprocessors (SMs), each of which consists of an array of streaming processors (SPs) or cores, which execute the threads [14]. Each core has its own ALU and a (shared) FPU. Data is stored in a memory hierarchy of thread-level registers, thread-level L1 cache, block-level shared memory, grid-level L2 cache, and off-chip device-level global, constant, and texture memories. The hardware capabilities of a device depend on its architecture version, which currently is one of Tesla (1.x compute capability), Fermi (2.x), or Kepler (3.x) architectures.

Our approach to accelerating a C code is to parallelize hot-spot functions by transforming an existing function into a host function that invokes a CUDA kernel derived from the function's core code. Since the CUDA model is based on extensions of C, the derivation of kernel code is based on a direct mapping from the existing C code. To ensure efficiency of the derived code we explored several optimizations, which we summarize next.

A. Optimizations

The NVIDIA GPU has different types of memory. Global memory is the largest; however, it also has the lowest bandwidth. Hence, strategies must be devised to keep global memory accesses to a minimum. Since Tesla devices have quite a few registers available per thread, an obvious first step is to look for per thread data reuse and explicitly move those operations to register storage. For the two Tesla GPUs we tested, the Telsa 1060 has 16,384 registers per device and each thread utilizes a maximum of 124, whereas the Tesla 2070 has 32,768 registers with a maximum register per thread of 63 [19].

Because the number of registers in use per thread is explicitly restricted, CUDA allows registers to spill over into its memory hierarchy. In the Fermi architecture, register spill proceeds through the cache hierarchy, unlike the Telsa 1060 where registers spill directly to the thread-local memory, located in the global memory [19].

Another strategy for avoiding and delaying global reads and writes is to offload values into shared memory. This is especially applicable to kernels that feature high data reuse across a thread block or must share data between threads within a block. Shared memory is limited, and its use can limit the size of a kernel launch; however, the performance gain from the use of shared memory can be significant compared with the use of global memory.

When global memory accesses are required in a kernel, those accesses must be coalesced into continuous reads or writes. On the Tesla 1060, shared-memory reads from global memory are executed per half-warp, that is, 16 threads at a time from a single warp of 32 threads. If reads are made from contiguous memory locations from global memory, they are performed concurrently through 16 hardware banks. Additionally, if all threads in a half-warp request access to a single global memory address, this request will also be broadcast to the entire half-warp in a single instruction. Memory accesses on the Tesla 1060 are processed in either 32, 64, or 128-byte segments. For coalesced accesses, the accesses to global memory must be aligned with these segments; otherwise, multiple accesses will be serialized.

Contrasting with the Tesla 1060, the Fermi architecture has 32 banks, allowing for a single global memory read to populate a full warp or to broadcast to a full warp from a single memory location. The Fermi architecture has a cache-based memory hierarchy, which relaxes the time penalty for uncoalesced and misaligned memory accesses. However, the techniques cited above are still important for increasing memory performance within a kernel.

B. OrCuda: Autotuner for CUDA

Orio provides an extensible framework for transformation and tuning of codes written in different source and target languages. Current support includes transformations from a number of simple languages (e.g., a restricted subset of C) to C and Fortran targets. We have extended Orio with transformations for CUDA, called OrCuda, where code written in restricted C is transformed into code in CUDA C.

Since loops take up a large portion of program execution time, our initial goal is to accelerate loops. In line with Orio's approach to tuning existing codes, we annotate existing C loops with transformation and tuning comments (specifications). Transformation specs drive the translation of annotated code into the target language code. The translated code is placed into a template `main` function. The tuning specs provide all the necessary parameters to build and execute an instance of the transformed code.

Transformations can be parametrized with respect to various performance-affecting factors, such as the size of a grid of thread blocks with which to execute a given CUDA kernel. Therefore, a transformation spec can generate a family of variant translations for each parameter. Each of the variants is measured for its overall execution time with the fastest chosen as the best-performing autotuned translation. This translation replaces the existing code to take full advantage of GPU acceleration.

Example: To illustrate our transformation and tuning approach, Figure 3 provides an example of annotated sparse matrix-vector multiplication, where the matrix A respresents a DIA-compression of the sparse 2dStar1-shaped Jacobian matrix of a two-dimensional grid. Here, the outer loop iterates over the rows and the inner loop iterates over the diagonals of the sparse matrix. The column index is based on an element's row index and the offset of the element's diagonal. If the column index is within the boundaries of the sparse matrix, then the corresponding elements of the matrix and the vector are multiplied and accumulated in the result vector. Note that for simplicity, here we use the standard DIA compression scheme as opposed to the extended DIA format without padding.

To accelerate this loop for execution on a CUDA-enabled GPU, we annotated it with a transformation and tuning specification. The transformation specs define a CUDA loop

```
void MatMult_SeqDia(double* A, double* x, double* y,
                    int m, int n, int nos, int dof) {
  int i,j,col;
  /*@ begin PerfTuning (
    def performance_params {
      param TC[] = range (32,1025,32);
      param BC[] = range (14,113,14);
      param UIF[] = range (1,6);
      param PL[] = [16,48];
      param CFLAGS[] = map(join,product(['','-use_fast_math'],
                                        ['','-O1','-O2','-O3']));
    }
    def input_params {
      param m[]     = [32,64,128,256,512];
      param n[]     = [32,64,128,256,512];
      param nos     = 5;
      param dof     = 1;
      constraint sq = (m==n);
    }
    def input_vars {
      decl static double A[m*n*nos*dof] = random;
      decl static double x[m*n*dof] = random;
      decl static double y[m*n*dof] = 0;
      decl static int offsets[nos] = {-m*dof,-dof,0,dof,m*dof};
    }
    def build {
      arg build_command = 'nvcc -arch=sm_20 @CFLAGS';
    }
    def performance_counter {
      arg repetitions = 5;
    }
  ) @*/
  int nrows=m*n;
  int ndiags=nos;
  /*@ begin Loop(transform CUDA(threadCount=TC, blockCount=BC,
    preferL1Size=PL, unrollInner=UIF)
  for(i=0; i<=nrows-1; i++) {
    for(j=0; j<=ndiags-1; j++){
      col = i+offsets[j];
      if(col>=0&&col<nrows)
        y[i] += A[i+j*nrows] * x[col];
    }
  }
  ) @*/
  for(i=0; i<=nrows-1; i++) {
    for(j=0; j<=ndiags-1; j++){
      col = i+offsets[j];
      if(col>=0&&col<nrows)
        y[i] += A[i+j*nrows] * x[col];
    }
  }
  /*@ end @*/
  /*@ end @*/
}
```

Fig. 3. Annotated DIA matrix-vector multiplication.

translation with parameterized transformation arguments for thread count, block count, and so forth. The body of the transformation spec contains unmodified C language code; however, this can be replaced by a higher-level (domain-specific) language code that captures salient computational features at a proper level of abstraction (e.g., stencil-based operations). Defining this and other domain languages and using them instead of the current C-based approach is part of planned future work.

The tuning specs provide machine- and device-specific parameters for instantiation of transformation variants, initialization of input variables used by transformed code, and the command for building an executable. Note that one can build the original code without performing any of the transformations—the annotation is nonintrusive to existing code and does not impact its portability. In this example, the performance of an executable will be averaged over five execution times. By default, for smaller examples such as this one, an exhaustive strategy is used, where all possible combinations of performance parameters are explored. Other search methods requiring fewer runs are also available in Orio. The highest-performing version replaces the annotated code in the final output of autotuning.

C. Host Function

In CUDA, the host function allocates memory on the device, transfer data from host memory to device memory, configures launch parameters of a kernel, and invokes the kernel. These activities are independent of the annotated source code that is being transformed (except when the data is already on the device) and vary only with respect to the data characteristics. OrCuda obtains the data sizes from the input variable section of the tuning specs. Next, OrCuda performs type inference and other analysis of the annotated code to identify scalars, arrays, types of identifiers and their uses and definitions. This information is used to generate CUDA API calls to allocate device memory of proper size and type, transfer the correct amount of data and pass appropriate parameters to a kernel function. For example, an excerpt of the generated host function for the example in Figure 3 is listed in Figure 4.

```
...
double *dev_y, *dev_A, *dev_x;
...
dim3 dimGrid, dimBlock;
dimBlock.x=32;
dimGrid.x=14;
cudaMalloc(&dev_y, m *n *dof*sizeof(double));
...
cudaDeviceSetCacheConfig(cudaFuncCachePreferL1);
...
cudaMemcpy(dev_y, y, m *n *dof*sizeof(double),
    cudaMemcpyHostToDevice);
...
orcu_kernel5<<<dimGrid,dimBlock>>>(nrows,ndiags,
   dev_offsets,dev_y,dev_A,dev_x);
...
cudaDeviceSetCacheConfig(cudaFuncCachePreferNone);
...
```

Fig. 4. Excerpt from an instance of a host function.

One of the important factors affecting the overall execution time of a kernel is its configuration [19]. This configuration includes specifications of how many threads are in a block and how many blocks are in a grid. OrCuda parameterizes these dimensions to enable a search for the best configuration. Transformation arguments *threadCount* and *blockCount* specify the dimensions of the grid of thread blocks. The tuning specs define the domains of these parameters as *range(l,u,i)*, which is a Python-based sequence $[l, u)$ in increments of i.

1) Thread count: The tuning specs vary from one device to another; however, we follow the general performance guidelines and technical specifications of CUDA architectures [19] in defining the search space. For example, all three existing architectures specify 32 as the size of a warp—the smallest

group of threads that can be scheduled for execution on an SM. Thus, the search space for thread counts starts at 32 in increments of 32. Based on the compute capability of a device, we can determine the upper bound, which is 512 for Tesla and 1024 for Fermi and Kepler architectures.

2) Block count: The scalability of the CUDA model derives from the notion of thread blocks—independently executing groups of threads that can be scheduled in any order across any number of SMs. While this restricts memory sharing across two blocks by disallowing interblock synchronization, it scales the acceleration with the capabilities of a device. The greater the number of SMs on a device, the greater the level of parallelism. For example, on a device that has an array of 14 SMs (Tesla C2070), up to 14 blocks can execute in parallel. Similarly, on a device with 30 SMs (Tesla C1060), up to 30 blocks can execute in parallel.

Technical specifications define the maximum number of blocks that can be resident on an SM at any time: 8 for Teslaand Fermi, 16 for Kepler architectures. Therefore, we define the search space of block counts as a multiple of device SMs starting from the minimum of the SM count up to maximum number of resident blocks. For example, the tuning spec in Figure 3 is configured for a Tesla C2070 Fermi device, which has 14 SMs.

3) Stream count: Another configuration feature that can improve acceleration is asynchronous concurrent execution via streams. Here, CUDA provides API functions that return immediately after the invocation and execute in a particular stream asynchronously to the host function or functions in other streams. This provides three types of concurrency:communication- hiding overlap of data transfer and kernel execution (deviceOverlap==1), concurrent execution of kernels (concurrentKernels==1) and even concurrent data transfers between the host and the device in both directions with overlapped kernel execution (asyncEngineCount==2). Support for each of these depends on the capability of a device indicated by the respective device property constraint.

OrCuda can query properties of a device using the CUDA API and determine whether the device supports stream-based concurrency. If streaming is supported (deviceOverlap && concurrentKernels), OrCuda divides the input data (when it is of uniform length) into equal-sized chunks and generates asynchronous data transfer calls. Then, it generates concurrent invocations of kernels to execute on a particular data chunk.

The transformation argument that controls streaming is *streamCount*. We define its domain as *range(1,17,1)*. When the count is one (default), OrCuda generates synchronous calls; for counts greater than one, it generates streaming calls. According to the CUDA specs, the maximum number of streams is 16, which is the upper bound of this parameter's domain.

Note that prior to a transformation, OrCuda performs a sanity check of the transformation arguments. If an argument's value is beyond the capabilities of a device, it raises an exception and does not perform the transformation. The tuning framework catches the exception and supplies the next combination of transformation argument values. This approach increases fault tolerance of the autotuning, ensuring that the search is not interrupted when the tuning specs contain invalid parameter ranges.

4) L1 size preference: On Fermi devices capable of caching global memory accesses, CUDA provides an API to toggle the size of the L1 cache. The same on-chip memory is used for L1 and block-level shared memory. One can set a preference to allocate 16 KB for L1 and 48 KB for shared memory (the default) or 48 KB for L1 and 16 KB for shared memory on Fermi devices. On Kepler devices, there is an additional configuration of 32 KB for L1 and 32 KB for shared memory (*cudaFuncCachePreferEqual*). A larger L1 cache can increase the performance of cache-starved kernels. Because this is just a preference, the CUDA runtime system ultimately decides whether to actually allocate the requested L1 size based on shared-memory requirements for a thread block.

OrCuda can generate the host-side API calls to set the preferred L1 size prior to the invocation of a kernel and to reset the preference to none after the invocation. Figure 4 illustrates an example of this capability.

5) Compiler flags: CUDA uses the `nvcc` compiler driver to generate PTX (assembly) code for further compilation into machine binaries. The *-arch=sm_xx* compiler option determines the compute capability when compiling kernel C code into PTX code. Other compiler flags can also be passed to `nvcc` to optimize the generated code. OrCuda uses the *@CFLAGS* build command parameter to specify various compiler option configurations for tuning. These configurations are generated by a Python-based expression for a cross-product of sequences of mutually exclusive options, which are then joined to form a single compiler option string. The tuning specs in Figure 3 provide an example of this functionality.

D. Device Functions

OrCuda transforms the annotated code and places the result into the body of a kernel function. All the identifiers used in the function become kernel parameters with a corresponding type. The primary transformation is the conversion of the outer loop into a loop executable by a single thread with the loop increment equal to the size of the grid of thread blocks. Figure 5 illustrates an example of a kernel function. Here, the thread ID is calculated based on the CUDA block and thread indices. Similarly, the grid size is based on the block and grid dimensions.

1) Reductions: OrCuda analyzes the annotated code to determine whether a loop performs an elementwise array update or array reduction. If it is not a reduction, the kernel consists of the transformed loop. Otherwise, the results of each thread are reduced within a thread block. If the input size is greater than a block's dimension, OrCuda generates a loop within the host function that performs cascading reductions across blocks.

Figure 5 illustrates the binary reduction, where threads in the first half of a block accumulate results of both halves. This

```
/* for (int i=0; i<=n-1; i++)
     r+=x[i]*y[i]; */
__global__ void orcu_kernel3(const int n, double* y, double* x,
     double* reducts) {
  const int tid=blockIdx.x*blockDim.x+threadIdx.x;
  const int gsize=gridDim.x*blockDim.x;
  __shared__ double shared_y[128];
  __shared__ double shared_x[128];
  double orcu_var5=0;
  for (int i=tid; i<=n-1; i+=gsize) {
    shared_y[threadIdx.x]=y[tid];
    shared_x[threadIdx.x]=x[tid];
    orcu_var5=orcu_var5+shared_x[threadIdx.x]*shared_y[threadIdx.x];
  }
  /*reduce single-thread results within a block*/
  __shared__ double orcu_vec6[128];
  orcu_vec6[threadIdx.x]=orcu_var5;
  __syncthreads();
  if (threadIdx.x<64)
    orcu_vec6[threadIdx.x]+=orcu_vec6[threadIdx.x+64];
  __syncthreads();
  if (threadIdx.x<32)
    orcu_warpReduce64(threadIdx.x,orcu_vec6);
  __syncthreads();
  if (threadIdx.x==0)
    reducts[blockIdx.x]=orcu_vec6[0];
}
__device__ void orcu_warpReduce64(int tid, volatile double* reducts) {
  reducts[tid]+=reducts[tid+32];
  reducts[tid]+=reducts[tid+16];
  reducts[tid]+=reducts[tid+8];
  reducts[tid]+=reducts[tid+4];
  reducts[tid]+=reducts[tid+2];
  reducts[tid]+=reducts[tid+1];
}
```

Fig. 5. An instance of a reduction kernel and a device function.

continues until only 64 elements are left to reduce, in which case a warp performs the last SIMD synchronous reduce [19].

2) Caching into shared memory: On-chip shared memory has substantially lower latency than does off-chip global memory [19]. On Tesla devices that do not have L1/L2 caches, caching data in shared memory can improve a kernel's performance. On devices with L1/L2 caches, caching into underutilized shared memory can also improve performance. OrCuda parameterizes the choice of caching into shared memory using the transformation argument *cacheBlocks*, which can have a Boolean value. Figure 5 illustrates a transformation variant when block-level caching is enabled (e.g., *shared_x* array).

3) Unrolling inner loops: To improve performance of kernels that contain inner loops, OrCuda generates a "`#pragma unroll n`" directive prior to the inner loop in order to indicate that the compiler should unroll the inner loop *n* times. Figure 3 illustrates an example of specifying transformation argument *unrollInner* with a range of $[1, 6)$, which corresponds to either no unrolling or unrolling up to the maximum number of five diagonals.

The transformations described here are an initial subset of the possible optimizations. We are exploring other general and domain-specific transformations to extend OrCuda.

IV. Performance Evaluation

Table I lists the two test platforms used in our performance experiments. All the source code is available from the repository at [1].

TABLE I
GPU platforms used for the evaluation.

GPU	Tesla C2070	Tesla C1060
Compute capability	2.0	1.3
Total number of cores	448	240
SM count	14	30
Clock rate	1147Mhz	1296Mhz
Global memory	5375MB	4095MB
Shared memory	48KB	16KB
Global memory bus width	384 bits	512 bits
Peak memory clock rate	1494MHz	800MHz
Number of registers	32768	16384
Max threads per block	1024	512
Max resident threads/SM	1536	1024
Number of async engines	2	1
L2 cache size	768KB	–

Table II lists the initial kernels we targeted for autotuning based on their use by the Krylov PETSc solvers. 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.

TABLE II
Kernel specifications.

Kernel	Operation
matVec	$y = Ax$
vecAXPY	$y = \alpha x + y$
vecMAXPY	$y = y + \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_n x_n$
vecDot	$w = x \cdot y$
vecNorm2	$\|x\|_2$
vecScale	$w = \alpha w$
vecWAXPY	$w = y + \alpha x$

We compare the performance of the kernels in Table II tuned with OrCuda with that of different library-based implementations. PETSc already includes vector and matrix types with GPU implementations that rely on CUSP [8] and Thrust [7]. While PETSc does not use cuBLAS [18], we use it as a baseline for comparison with the different vector operation implementations because it is the best-performing among the available library options.

Figure 6 shows the execution times of the tuned CUDA code computing dense vector 2-norm and dot product for three different vector sizes (10^5, 10^6, and 10^7), normalized by the kernel times for the corresponding cuBLAS implementations. Here, in addition to the CUSP and cuBLAS versions, we include our own hand-tuned custom versions. In all cases for both devices, the autotuned kernels outperform the other versions.

Figure 7 shows the execution times of the tuned CUDA code for other dense vector operations for vector sizes (10^5, 10^6, and 10^7), normalized by the kernel times for the corresponding cuBLAS implementations. As before, autotuned versions outperform others except for the vecMAXPY kernel on a Tesla device, which is explained by the lack of L1/L2 caches on these devices.

Figure 8 shows the execution times of the tuned CUDA code for matrix-vector product and the corresponding cuSPARSE

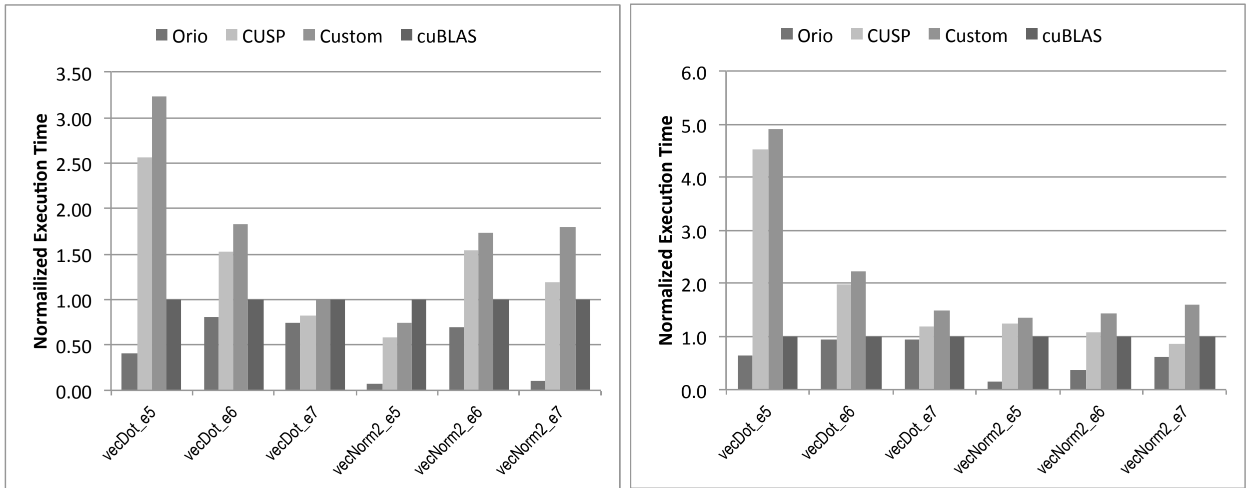

Fig. 6. Execution time for reduction kernels, normalized by the cuBLAS time (equal to 1.0 in these plots) on Fermi C2070 (left) and Tesla C1060 (right).

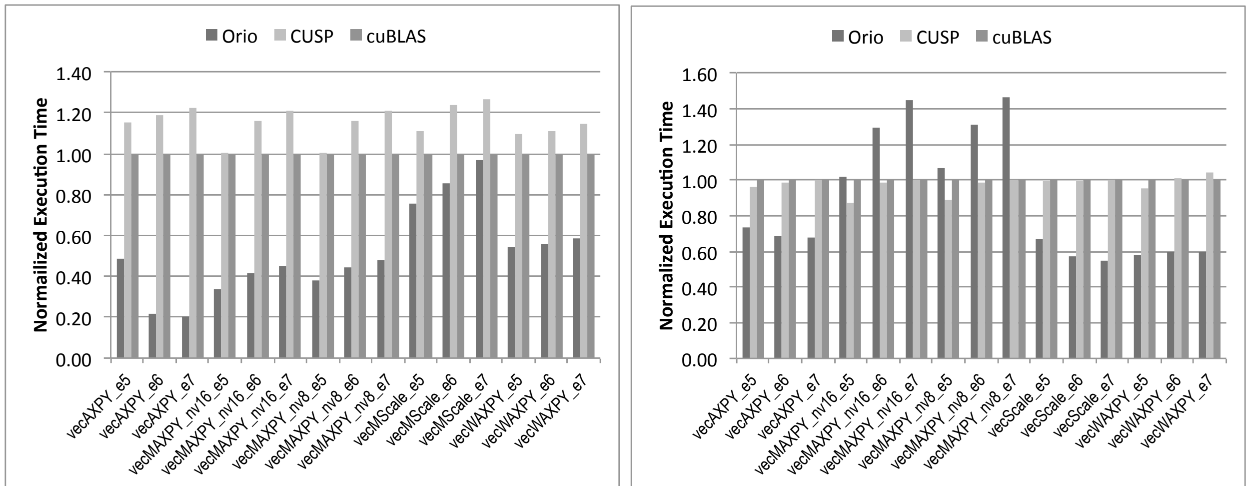

Fig. 7. Execution time for vector operation kernels, normalized by the cuBLAS time (equal to 1.0 in these plots) on Fermi C2070 (left) and Tesla C1060 (right).

CSR function for matrices corresponding to a 5-point stencil discretization for several 2-D and 3-D grid sizes. Similar to the previous experiment, the autotuned versions outperform the other versions on devices with a cache hierarchy.

V. Related Work

Libraries such as CUSP [8], [6], Thrust [7], and cuBLAS [18] provide optimized CUDA implementations of many numerical kernels used in scientific computing. These implementations, however, are not tunable for specific problem characteristics. Furthermore, one cannot take advantage of spatial and temporal locality in multiple consecutive operations on the same matrices or vectors.

The MAGMA project [21], [15] aims to develop a library similar to LAPACK [2], but for heterogeneous architectures, initially focusing on CPU+GPU systems. MAGMA supports dense (full) matrix algebra, unlike our approach, which focuses on sparse matrix algebra in support of stencil-based PDE discretizations.

Other autotuning systems are also beginning to target hybrid architectures. For example, the combination of the CHiLL and ActiveHarmony tools can process C code and empirically tune the generated CUDA code [12], [20]. The goals of this approach are similar to ours. Because the existing CPU code itself is used as input, the complexity of the CPU implementation may prevent the optimization of CUDA code. Unlike our domain-specific approach, this more general approach makes it harder to exploit domain-specific properties, such as the regular structure of the stencil-based matrices.

VI. Conclusions and Future Work

We have described our initial implementation of support for CUDA code generation and autotuning for a set of key numerical kernels in PETSc. The performance of the autotuned implementations often exceeds that of optimized vendor libraries for different problem sizes. While here we present results for individual kernels, our ultimate goal is to enable autotuning at the linear solver algorithm level. This

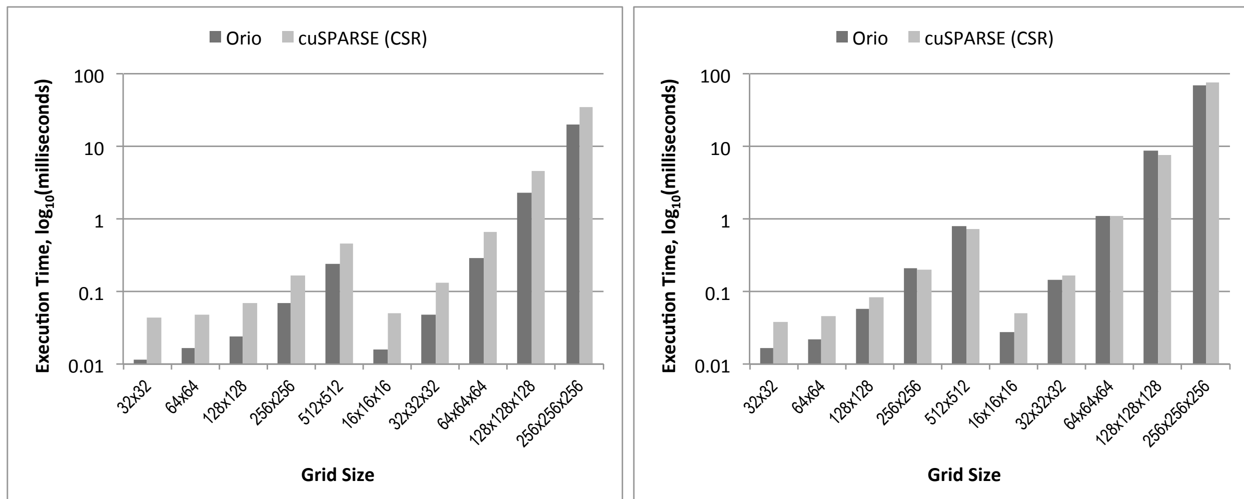

Fig. 8. Execution time for matrix-vector product for matrices corresponding to the stencil-based discretization of several grid sizes on Fermi C2070 (left) and Tesla C1060 (right).

will present even greater optimization opportunities because we will be operating on a larger set of vector and matrix operations. In addition to the solver implementation in PETSc, we are developing autotuning support for the application-specific stencil update functions (i.e., `FormFunction` and `FormJacobian` in PETSc).

ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy Office of Science under Contract No. DE-AC02-06CH11357.

REFERENCES

[1] Orio Project. http://tinyurl.com/OrioTool, 2008.
[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, second edition, 1995.
[3] S. Balay, J. Brown, , K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.2, Argonne National Laboratory, 2011.
[4] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2011. http://www.mcs.anl.gov/petsc.
[5] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.
[6] N. Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2012. Version 0.3.0.
[7] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. In W. mei W. Hwu, editor, *GPU Computing Gems, Jade Edition*. Oct. 2011.
[8] R. Galvez and G. van Anders. Accelerating the solution of families of shifted linear systems with CUDA. http://arxiv.org/abs/1102.2143, 2011.
[9] J. Godwin, J. Holewinski, and P. Sadayappan. High-performance sparse matrix-vector multiplication on gpus for structured grid computations. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 47–56, New York, NY, USA, 2012. ACM.
[10] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. K. namoorth, B. Norris, J. Ramanujam, and P. Sadayappan. PrimeTile: A parametric multi-level tiler for imperfect loop nests. In *Proceedings of the 23rd International Conference on Supercomputing*, IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, June 2009.
[11] Hypre. http://acts.nersc.gov/hypre/, 2008.
[12] M. Khan, J. Chame, G. Rudy, C. Chen, M. Hall, and M. Hall. Automatic high-performance GPU code generation using CUDA-CHiLL, 2011. poster.
[13] X. S. Li and J. W. Demmel. SuperLU DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.
[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.
[15] R. Nath, S. Tomov, and J. Dongarra. An improved magma gemm for fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.
[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, March 2008.
[17] B. Norris, A. Hartono, E. Jessup, and J. Siek. Generating empirically optimized composed matrix kernels from MATLAB prototypes. In *Proceedings of the International Conference on Computational Science 2009*, 2009. Also available as Preprint ANL/MCS-P1581-0209.
[18] NVIDIA. NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) Library. http://developer.nvidia.com/cublas, 2012. Last accessed April 28, 2012.
[19] NVIDIA. NVIDIA CUDA C Programming Guide Version 4.2, 2012.
[20] G. Rudy. *CUDA-CHiLL: A Programming Language Interface for GPGPU Optimizations and Code Generation*. PhD thesis, The University of Utah, Aug. 2010.
[21] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5–6):232–240, 2010.
[22] Trilinos. http://software.sandia.gov/trilinos, 2008.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.