STENCIL-AWARE GPU OPTIMIZATION OF ITERATIVE SOLVERS[*]

CHEKURI CHOUDARY,[3] JESWIN GODWIN,[2] JUSTIN HOLEWINSKI,[2]
DEEPAN KARTHIK,[2] DANIEL LOWELL,[1] AZAMAT MAMETJANOV,[1]
BOYANA NORRIS,[1] GERALD SABIN,[3] P. SADAYAPPAN[2]

[1]MATHEMATICS AND COMPUTER SCIENCE DIVISION
ARGONNE NATIONAL LABORATORY
ARGONNE, IL 60439
[DLOWELL,AZAMAT,NORRIS]@MCS.ANL.GOV

[2]DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
OHIO STATE UNIVERSITY
COLUMBUS, OH 43210
[GODWIN,HOLEWINS,BALASUBD,SADAY]@CSE.OHIO-STATE.EDU

[3]RNET TECHNOLOGIES, INC.
DAYTON, OH 45459
[CCHOUDARY,GSABIN]@RNET-TECH.COM

Abstract. Numerical solutions of nonlinear partial differential equations frequently rely on iterative Newton-Krylov methods, which linearize a finite-difference stencil-based discretization of a problem, producing a sparse matrix with regular structure. Knowledge of this structure can be used to exploit parallelism and locality of reference on modern cache-based multi- and many-core architectures, achieving high performance for computations underlying commonly used iterative linear solvers. In this paper we describe our approach to sparse matrix data structure design and our implementation of the kernels underlying iterative linear solvers in PETSc. We also describe autotuning of CUDA implementations based on high-level descriptions of the stencil-based matrix and vector operations.

Key words. structured grid, sparse matrix format, iterative solvers, autotuning, GPGPU, PETSc

AMS subject classifications. 65Y10, 65F50, 15A06, 68N19

1. Introduction.

Many scientific applications rely on high-performance numerical libraries, such as Hypre [17], PETSc [5–7], SuperLU [19], and Trilinos [27], for providing accurate and fast solutions to problems modeled by using nonlinear partial differential equations (PDEs). Thus, the bulk of the burden in achieving good performance and portability is placed on the library implementors, largely freeing computational scientists from low-level performance optimization and portability concerns. At the same time, the increasing availability of hybrid CPU/accelerator architectures is making the task of providing both portability and high performance in both libraries and applications increasingly challenging. The latest Top500 list [2] contains thirty-nine supercomputing systems with GPGPUs. Amazon has announced the availability of Cluster GPU Instances for Amazon EC2. More and more researchers have access to GPU clusters instead of CPU clusters for large-scale computation problems in areas such as high energy physics, scientific simulation, data mining, climate forecast, and earthquake prediction. Relying entirely on compilers for code optimization does not produce satisfactory results, in part because the languages in which libraries are implemented (C, C++, Fortran) fail to expose sufficient information required for aggressive optimizations, and in part because of the tension between software design and performance—a well-engineered, dynamically extensible library is typically much more difficult to optimize through traditional compiler approaches.

[*]This work builds on and significantly extends previous work by the authors described in [15,20].

Our goal is to tackle the challenges in achieving the best possible performance on hybrid CPU/ GPGPU architectures at the library level by exploiting known problem structure and algorithmic properties. Unlike methods that focus exclusively on the algorithms and their implementations, our approach considers both the principal data structures and the operations on them. We focus on structured grid applications and PETSc, a widely used library for the nonlinear solution of PDE-based problems. Performance is typically dominated by the linear solution; hence, we consider the sparse matrix and vector data types and associated operations.

Our contributions can be summarized as follows.

- A structured grid-based matrix data structure that facilitates SIMD parallelism better than general sparse matrix formats
- Manually optimized matrix and vector kernel implementations
- Autotuning support in Orio for CUDA code optimization based on high-level, simple definitions of the matrix and vector kernels used in iterative linear solvers in PETSc
- Entire PETSc-based application execution on the GPGPU, including application-specific functions

The rest of the paper is organized as follows. Section 2 describes existing sparse matrix formats and the software packages we use and extend for this work. Section 3 presents our technical approach to the design of a new stencil-based matrix data structure and the implementation and autotuning of key matrix and vector kernels on the GPU. In Section 4 we present kernel performance results, as well as a comparison of GPU and CPU performance for a complete application. In Section 5 we review related work and conclude with a summary and future work description in Section 6.

2. Background.

We begin our discussion with a brief contextual overview of storage formats and software packages used in this work.

2.1. Sparse Matrix Storage Formats.

Bell and Harland [8] proposed several sparse matrix storage formats, each optimized for different use cases. In this section, we briefly describe three prominent formats: compressed sparse row (CSR), blocked CSR, diagonal (DIA), and ELLPACK. Figure 2.1 shows these different formats. The libraries for linear algebraic computations and numerical simulations support other formats as well. For example, the PETSc library supports CSR, blocked CSR, and a few other sparse matrix formats. The Cusp library supports CSR, DIA, ELLPACK, and some other formats.

2.1.1. Compressed Sparse Row Format.

CSR is a matrix storage technique that aims to minimize the storage for arbitrary sparse matrices. In this format, the nonzero elements in each row of the sparse matrix are stored contiguously. The rows containing only nonzero elements are stored in a single dense array (A). Two additional vectors are used to track the column index of each nonzero element (J) and the offset into A of the start of each row. The storage complexity for CSR format is on the order of $O(2N_{nz} + N_r + 1)$, where N_{nz} is the total number of nonzeros and N_r is the number of rows. The CSR format requires indirection for accessing matrix elements during matrix operations.

2.1.2. Blocked CSR.

The blocked CSR representation exploits the blocking structure that arises in certain applications. The storage format is similar to CSR except that it is a blocked version. Instead of storing nonzero elements contiguously, small two dimensional blocks with at least one nonzero element are flattened and stored contiguously. Each block is padded with zeros to fill the block and is stored

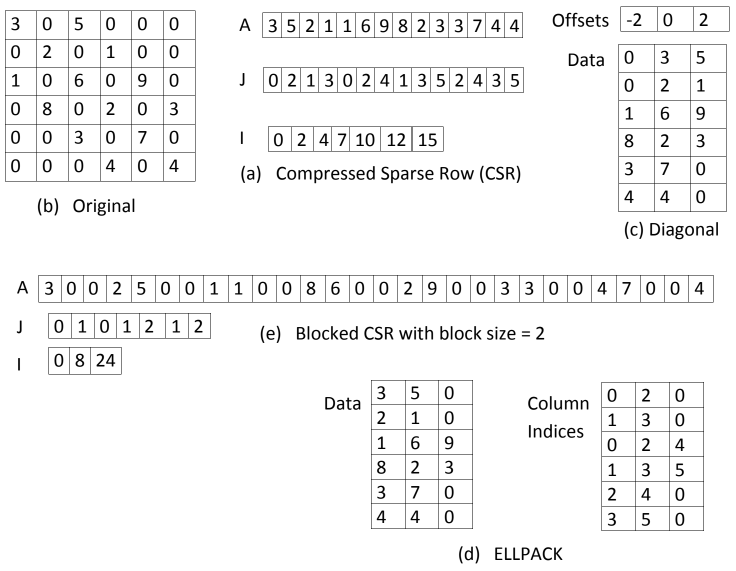


Fig. 2.1: Examples of common sparse matrix formats.

in a row major format. The additional vectors store the block column indices and the offset into the start of each block row. The storage complexity for blocked CSR format is on the order of $O((N_b * size) + N_b + (N_r/size))$, where N_b is the total number of blocks containing at least one nonzero element, $size$ is the block size, and N_r is the number of rows. The advantages of blocked CSR format are register blocking, reduced indirection compared to CSR, and reduced storage space. The optimal block size is matrix dependent and machine dependent and is usually obtained by using a performance model [28].

2.1.3. Diagonal Format. DIA is specifically suitable for storing sparse matrices that contain nonzero elements only along the matrix diagonals. In this format, the diagonals with nonzero elements are laid out contiguously in a dense matrix structure (data), starting with the lowest subdiagonal and ending with the highest superdiagonal. An additional vector stores the offset of each diagonal from the central diagonal. The storage space for the diagonal format is on the order of $O((N_d * N) + N_d)$, where N_d is the number of occupied diagonals and N is the width of the matrix.

2.1.4. ELLPACK. The ELLPACK [1] format offers an efficient storage format if the maximum number of nonzero elements in any row is significantly less than the number of columns in the sparse matrix. If K is the maximum number of nonzero elements in any row of a sparse matrix containing N_r rows, it is stored as an $N \times K$ matrix. Each row of this matrix contains contiguously stored nonzero elements of the corresponding row in the sparse matrix. Each row is padded with zeros to contain K elements. The column indices for elements in each row are stored as another $N \times K$ matrix. Consequently, the storage complexity is $O(2 * N * K)$.

2.2. Software packages. This work relies on and extends two software packages: the Portable, Extensible Toolkit for Scientific Computation (PETSc) and the autotuning framework Orio.

2.2.1. PETSc. PETSc [5–7] is an object-oriented toolkit for the numerical solution of nonlinear PDEs. Solver algorithms and 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 $f : R^n \rightarrow R^n$, 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.

2.2.2. 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 showed that high-level computation specifications can be embedded in existing C or Fortran codes by expressing them through annotations specified as structured comments [16, 22], as illustrated in Figure 2.2. 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. In this work, we describe an extension of Orio for transforming existing C code into CUDA code and tuning it for different GPU architectures.

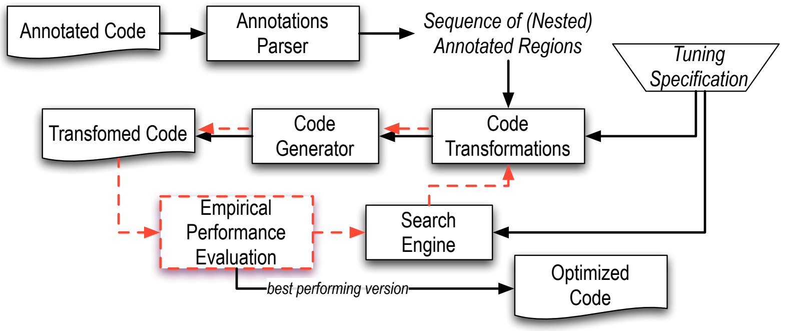


Fig. 2.2: Orio autotuning process.

3. Approach. Our approach is a comprehensive strategy for exploiting the known structure in finite-difference, discretization-based PDE solutions on regular grids, by rethinking the key data structure and autotuning the kernels that typically dominate the runtime of large-scale applications involving nonlinear PDE solutions using finite-difference or finite-volume approximations. Many other works (e.g., [11, 25, 29]) explore this structure, but address only a portion of the relevant kernels and typically rely on general sparse matrix formats such as those described in Section 2.1.

Structured grids are a key computational pattern in scientific computing that approximates the solution of a differential equation by discretizing the problem domain on a structured grid and computing the differences of the model function values at neighboring grid points 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 time step 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 the stencil $s \in \{star, box\}$, and the stencil's width $w \in N$. For example, given a star-shaped stencil of width 1 applied to a two-dimensional domain, each grid point interacts with four of its neighbors to the left, right, above, and below its position within the grid.

Many applications, such as those based on finite-difference and finite-volume formulations, use stencil-based structured grid computations. For explicit formulations, the stencil function is applied to each element in the grid explicitly. For solvers based on implicit formulations, however linear systems of equations arise, where the sparsity pattern of the matrix or the linear systems bears a direct relationship to a regular stencil around each grid point. For such applications, the compute performance of the application is typically dominated by the linear solution.

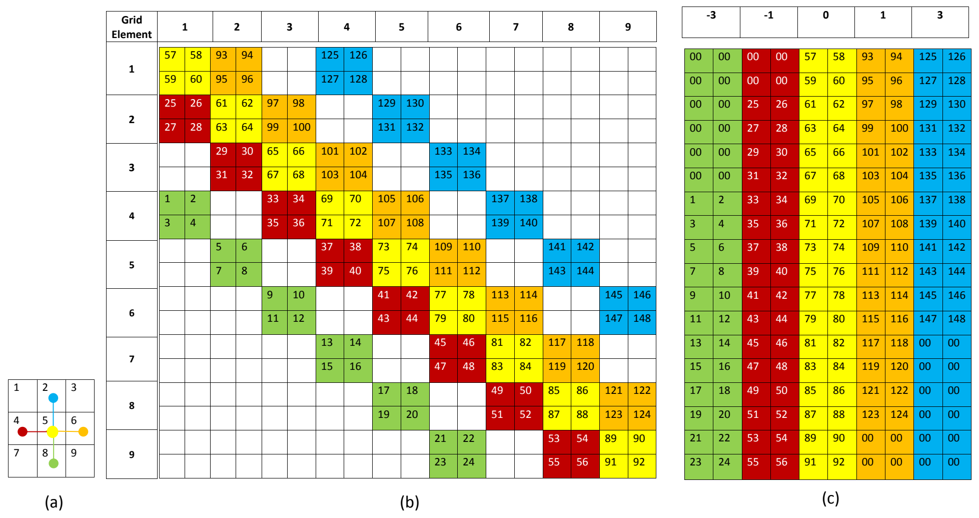
An interesting case arises when we consider structured grids with higher dimensional entities. Many applications use vector quantities at grid points and matrices as coefficients that relate the grid points. The dimension of the vector at each grid point is the number of degrees of freedom for the application. For such problems, the storage format used for sparse matrix plays a significant role in the memory efficiency and the performance of computational kernels such as matrix-vector multiplication that arise in the iterative linear solver.

3.1. Structured Grid-Diagonal Format. Optimizing algorithms without considering the data structures limits the types and effectiveness of possible optimizations. Hence, we begin by designing the sparse data structure for representing the linear system in order to exploit the known structure and enable efficient parallelization and memory access in the key computational kernels underlying Krylov iterative solvers. The following disadvantages of sparse matrix formats commonly used for structured grid computations on GPGPUs motivate our development of a new sparse matrix format.

1. The CSR format requires indirection for both loop bounds and vector indexing. This indirection prevents most compiler optimizations. Moreover, SIMD parallelism is constrained across the rows of the matrix.
2. The CSR format does not consider the diagonally dominant structure arising in structured grid problems. This results in discontiguous memory accesses to the input vector in sparse matrix-vector multiplication (SpMV).
3. The blocked CSR format provides reuse of the input vector in SpMV but still suffers from the other disadvantages of CSR format.
4. The diagonal format requires extra zeros to fill the gaps in the diagonals and hence results in increased memory bandwidth requirements and more floating-point operations.

We introduce a specialized data structure called *structured grid-diagonal* format (SG-DIA) for stencil-based structured grid computations. It is based on extensive experiments with linear systems arising in structured grid problems with degrees of freedom greater than or equal to one.

Figure 3.1(a) shows a 3×3 grid with an overlayed 5-point stencil. If a 2x1 vector quantity is considered at each grid point (i.e., two degrees of freedom), the structure of the Jacobian in a typical numerical simulation would be as shown in Figure 3.1(b). Each grid element in the mesh bears a direct relationship to a set of two rows in this matrix. The matrix also contains entries for neighbors under Dirichlet or periodic boundary conditions. The stencil function and the multiple degrees of freedom at each grid point result in a blocked diagonal structure. We store this matrix based on its number of degrees of freedom. Each blocked diagonal in the original matrix becomes a pair of columns in our SG-DIA format. These columns are laid out as shown in Figure 3.1(c). The layout is stored in a column-major format. The SG-DIA format further encodes the offsets to the blocked diagonals from the main diagonal as shown at the top of Figure 3.1(c). The blocked diagonals, the offsets, and the number of degrees of freedom provide sufficient information to determine the positions of the individual elements in the matrix.



Grid Element	1		2		3		4		5		6		7		8		9	
1	57	58	93	94			125	126										
	59	60	95	96			127	128										
2	25	26	61	62	97	98			129	130								
	27	28	63	64	99	100			131	132								
3			29	30	65	66	101	102			133	134						
			31	32	67	68	103	104			135	136						
4	1	2			33	34	69	70	105	106			137	138				
	3	4			35	36	71	72	107	108			139	140				
5			5	6			37	38	73	74	109	110			141	142		
			7	8			39	40	75	76	111	112			143	144		
6					9	10			41	42	77	78	113	114			145	146
					11	12			43	44	79	80	115	116			147	148
7							13	14			45	46	81	82	117	118		
							15	16			47	48	83	84	119	120		
8									17	18			49	50	85	86	121	122
									19	20			51	52	87	88	123	124
9											21	22			53	54	89	90
											23	24			55	56	91	92

-3		-1		0		1		3	
00	00	00	00	57	58	93	94	125	126
00	00	00	00	59	60	95	96	127	128
00	00	25	26	61	62	97	98	129	130
00	00	27	28	63	64	99	100	131	132
00	00	29	30	65	66	101	102	133	134
00	00	31	32	67	68	103	104	135	136
1	2	33	34	69	70	105	106	137	138
3	4	35	36	71	72	107	108	139	140
5	6	37	38	73	74	109	110	141	142
7	8	39	40	75	76	111	112	143	144
9	10	41	42	77	78	113	114	145	146
11	12	43	44	79	80	115	116	147	148
13	14	45	46	81	82	117	118	00	00
15	16	47	48	83	84	119	120	00	00
17	18	49	50	85	86	121	122	00	00
19	20	51	52	87	88	123	124	00	00
21	22	53	54	89	90	00	00	00	00
23	24	55	56	91	92	00	00	00	00

Fig. 3.1: Specialized data structure for stencil-based structured grid computations. (a) Sample structured grid with overlayed stencil. (b) Sparsity pattern of the Jacobian. (c) The SG-DIA storage format.

For a problem with grid dimensions $m \times n \times p$, degrees of freedom d_f, and a k-point star-type stencil function, the matrix contains k blocked diagonals. For example, in Figure 3.1(b), there are five blocked diagonals for five stencil points, each color-coded separately. In the SG-DIA format, the columns resulting from each blocked diagonal are padded with zeros such that the height of each column is $mnpd_f$. The width of the matrix is kd_f and the total number of matrix entries stored is $mnpkd_f^2$. We also store the offsets for each of the k blocked diagonals. Hence, the total number of entries stored by SG-DIA is $mnpkd_f^2 + k \approx mnpkd_f^2$.

The number of entries required to store the matrix arising in this problem in the traditional diagonal format (DIA, see Section 2.1.3) is calculated as follows. Each blocked diagonal resulting from a stencil point corresponds to $(2d_f - 1)$ diagonals in the DIA format. For example, in Figure 3.1(b) the green blocked diagonal resulting from two degrees of freedom corresponds to three diagonals in DIA format. However, since the mesh elements are numbered sequentially in one of the three dimensions, $\lfloor \frac{k}{2} \rfloor + 1$ blocked diagonals are adjacent to one another (red, light yellow, and dark yellow blocked diagonals in Figure 3.1(b) correspond to seven diagonals in DIA format). Hence, in order to store the matrix in DIA format, the total number of diagonals is $k(2d_f - 1) - \lfloor \frac{k}{2} \rfloor$. Each diagonal requiring storage is padded with zeros to a length of $mnpd_f$. In addition, $k(2d_f - 1) - \lfloor \frac{k}{2} \rfloor$ offsets for the diagonals are stored. Hence, the number of elements stored by DIA is $mnpkd_f(2d_f - \frac{3}{2}) + k(2d_f - 1) - \lfloor \frac{k}{2} \rfloor \approx 2(mnpkd_f^2)$, or roughly twice the memory used by the new SG-DIA format.

For CSR format, for the sake of simplicity (and without introducing significant additional nonzeros for most problem sizes), assume zero-padding of some of the rows to store kd_f elements per row. The matrix has $mnpd_f$ rows, so the total number of stored elements is $2mnpkd_f^2 + mnpd_f + 1$, or about twice the memory used by SG-DIA. The SG-DIA and DIA formats avoid the column indices and the row offsets that are required in CSR format. In addition, the SG-DIA format stores only the offsets for the blocked diagonals, which is the number of stencil points k, whereas the DIA format stores the offsets for all the diagonals, which is $\approx 2kd_f$.

Assuming 32-bit storage for matrix entries and the indices, SG-DIA uses approximately half the storage space that the DIA uses. The conservation of storage space has the following advantages on GPU accelerators. First, we require less total memory to store the sparse matrix. Modern GPU accelerators have up to 6 GB of off-chip storage, whereas high-performance CPUs typically have 32-64 GB. The matrix operations for large problems require multiple kernel launches that consume a significant amount of time in GPU applications. Memory conservation reduces the number of kernel launches for such large problems (that do not entirely fit in the GPU memory). Second, less data needs to be transferred to the GPU. Data transfer cost is a major bottleneck for GPU acceleration. Third, eliminating the extra zeros in the DIA format reduces the number of floating-point operations in matrix computations. For matrix-vector multiplication, the number of floating-point operations is twice the number of stored matrix entries (multiply-add for each entry). By reducing the number of stored zeros compared with that of the DIA format, we approximately halve the floating-point operations in SpMV.

Sparse Matrix-Vector Multiplication on GPUs. We developed a CUDA implementation for sparse matrix-vector multiplication (SpMV: $\vec{y} = A\vec{x}$) based on our SG-DIA storage format shown in Figure 3.1(c). The pseudocode for this implementation is shown in Algorithm 1. The parameters A, x, and y are the input/output matrix and vectors; VecSize is the size of the vectors in number of elements $(m * n * p * d_f)$; NumDiags is the number of block diagonals (i.e., stencil points); DiagOffsets is the offsets to the blocked diagonals; and DOF is the number of degrees of freedom. Each GPU thread processes one element of the output vector $\vec{y}$.

Each thread computes the dot product between the vector $\vec{x}$ and a row of the matrix A. The sparsity pattern of the matrix allows us to form this dot product by considering at most DOF $*$ NumDiags columns. Lines 1–3 initialize the kernel by determining the ID of the current thread (implemented by the `GetGlobalThreadId()` function), zeroing the summation variable, and precomputing the size, in elements, of each block diagonal.

The outer loop at line 4 processes each block diagonal. Lines 5–6 compute the base offsets into the sparse matrix data structure and vector $\vec{x}$. The inner loop at line 7 processes each column that falls into the current block diagonal. In our SG-DIA sparse matrix representation, elements in the same row are offset by VecSize elements; thus the i^{th} element will be at offset VecSize$*i$. The `TexSample(t, i)` function accesses t[i] using the texture memory hardware in the GPU device. The `ysum` variable is used as the summation variable and is written to the correct element of $\vec{y}$.

Algorithm 1: Pseudocode for the SpMV kernel.

```
   Parameters: A, x, y, VecSize, NumDiags, DiagOffsets, DOF
 1 idx ← GetGlobalThreadId ()
 2 ysum ← 0
 3 diag_size ← VecSize ∗ DOF
 4 foreach $j \in [0, \text{NumDiags} - 1]$ do
 5     d ← DiagOffsets[j]
 6     offset ← diag_size ∗ $j$ + idx
 7     xoff ← $(\lfloor \frac{\text{idx}}{\text{DOF}} \rfloor + d) * \text{DOF}$
 8     foreach $i \in [0, \text{DOF} - 1]$ do
 9         coeff ← A[offset + VecSize ∗ $i$]
10         xval ← TexSample (x, xoff + $i$)
11         ysum ← ysum + coeff ∗ xval
12     end
13 end
14 y [idx] ← ysum
```

3.2. Autotuning. Our initial experience with the manually implemented SpMV using the new data structure served as the basis for extending Orio with CUDA code generation and optimization capabilities. As described in Section 2.2.2, Orio provides an extensible framework for transformation and tuning of codes written in different source and target languages, including 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 simple, restricted C is transformed into code in CUDA C.

Because kernels containing computation-intensive loops dominate execution time in the applications we are targeting, accelerating these kernels is a reasonable first step. Similar to the approach we use for CPU optimizations, 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, which can be compiled and executed in order to evaluate its performance.

Transformations can be parameterized 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, we provide in Figure 3.2 an example of annotated sparse matrix-vector multiplication, where the

```
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. 3.2: Annotated matrix-vector multiplication.

matrix A respresents a SG-DIA compression of the sparse Jacobian matrix resulting from applying a 2-D 5-point stencil on a 2-D grid for a problem with one degree of freedom per grid point. Note that the longer comment containing the tuning spec (the *PerfTuning* annotation) can be specified in a separate file and need not be embedded in the source code. Here, the outer loop iterates over the rows, and the inner loop iterates over the diagonals of the sparse matrix (the innermost loop is omitted because this example considers d_f=1). 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.

To generate efficient CUDA code with Orio, we annotated the code with a transformation and tuning specification. The transformation specs define a CUDA loop 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 are 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 [4,16]. The highest-performing version replaces the annotated code in the final output of autotuning. Orio also optionally performs validation of tuned kernels by comparing their results with those of the original (not tuned) kernel.

3.2.1. Host Function. In CUDA, the host function allocates memory on the device, transfers 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. An excerpt of the generated host function for the example in Figure 3.2 is listed in Figure 3.3.

```
...
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. 3.3: Excerpt from an instance of a host function.

One of the important factors affecting the overall execution time of a kernel is its configuration [24]. This configuration includes specifications of how many threads are in a block and how many blocks are in a grid. OrCuda parameterizes these dimensions in order 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.

3.2.2. Thread count. The tuning specs vary from one device to another; however, we follow the general performance guidelines and technical specifications of CUDA architectures [24] 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 a streaming multiprocessor (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: 512 for Tesla and 1024 for Fermi and Kepler architectures.

3.2.3. 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 Tesla and 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 3.2 is configured for a Tesla C2070 Fermi device, which has 14 SMs.

3.2.4. 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 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 by using the CUDA API and can 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, OrCuda 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.

3.2.5. 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 3.3 illustrates an example of this capability.

3.2.6. 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` in order 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 3.2 provide an example of this functionality.

3.2.7. 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 3.4 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.

3.2.8. 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 3.4 illustrates the binary reduction, where threads in the first half of a block accumulate results of both halves. This continues until only 64 elements are left to reduce, in which case a warp performs the last SIMD synchronous reduce [24].

```
/* 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. 3.4: Instance of a reduction kernel and a device function.

3.2.9. Caching into shared memory. On-chip shared memory has substantially lower latency than does off-chip global memory [24]. 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 by using the transformation argument *cacheBlocks*, which can have a Boolean value. Figure 3.4 illustrates a transformation variant when block-level caching is enabled (e.g., *shared_x* array).

3.2.10. 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 3.2 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.

4. Performance Evaluation. Table 4.1 lists the platforms used for evaluating the performance of the new sparse matrix type and associated manually implemented and autotuned kernels. The development and initial testing of the sparse matrix

Table 4.1: GPU platforms used for evaluation (the devices marked with * were used for autotuning).

	Quadroplex S2200	**Tesla C1060***	**Tesla C2050**	**Tesla M2070**	**Tesla C2070***
Compute Capability	1.3	1.3	2.0	2.0	2.0
CUDA Cores	240	240	448	448	448
SM Count	30	30	14	14	14
Clock Rate	1.30 GHz	1.30 GHz	1.15 Ghz	1.15 GHz	1.15 GHz
Global Memory	4,096 MB	4,095 MB	2,687 MB	5,375 MB	5,375 MB
Shared Memory	16 KB	16 KB	48 KB	48 KB	48 KB
Memory Bus Width	512-bit	512-bit	384-bit	384-bit	384-bit
Peak Memory Clock	800 MHz	800 MHz	1,500 MHz	1,566 MHz	1,494 MHz
Registers/SM	16,384	16,384	32,768	32,768	32,768
Max Threads/Block	512	512	1,024	1,024	1,024
Max Threads/SM	1,024	1,024	1,536	1,536	1,536
L2 Cache Size	–	–	786 KB	786 KB	768 KB

data structure were performed on the Quadroplex S2200 and Tesla C2050 and M2070 because these were the GPUs most readily available to the portion of the team who was primarily responsible for the data structure design. The autotuning experiments take a significant amount of time and are best performed on a dedicated system; hence, we performed the tuning on the C1060 and C2070 cards, which were most readily available to the autotuning team members and are identical or similar to the Quadroplex S2200 and M2070, respectively. The code was compiled with Intel v. 11 compilers and nvcc 4.2.

4.1. Kernel Performance. To study the effects of different degrees of freedom, we evaluate the performance of the SG-DIA storage format and the matrix-vector product kernel (SpMV) implementation described in Section 3.1 on three NVIDIA GPU architectures: Quadroplex S2200, Tesla C2050, and Tesla M2070. We generate test matrices with sparsity patterns that arise in structured grid applications with multiple degrees of freedom. We compare the performance of the SG-DIA format with different sparse matrix storage formats available in the Cusp library. Figure 4.1 shows the performance of the SG-DIA format for two grid sizes compared with Cusp implementations of other formats for both 32-bit single-precision floating-point numbers and 64-bit double-precision floating-point numbers. As shown in the figure, our kernel achieves higher performance compared with that of all other matrix formats as we increase the number of degrees of freedom in the problem.

Table 4.2 lists the initial set of kernels we targeted for autotuning based on their use by the Krylov PETSc solvers. The operation notation is as follows: A designates a matrix; $x, x_1, \ldots, x_n$, y, and w are vectors; and $\alpha, \alpha_1, \ldots, \alpha_n$ are scalars.

We compare the performance of the kernels in Table 4.2 tuned with OrCuda with that of different library-based implementations. PETSc already includes vector and matrix types with GPU implementations that rely on Cusp [13] and Thrust [10]. While PETSc does not use cuBLAS [23], 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 4.2 shows the execution times of the autotuned CUDA code computing dense vector 2-norm and dot product for three vector sizes (10^5, 10^6, and 10^7), normalized by the kernel times for the corresponding cuBLAS implementations. In all cases for both devices, the autotuned kernels outperform the other versions. A more

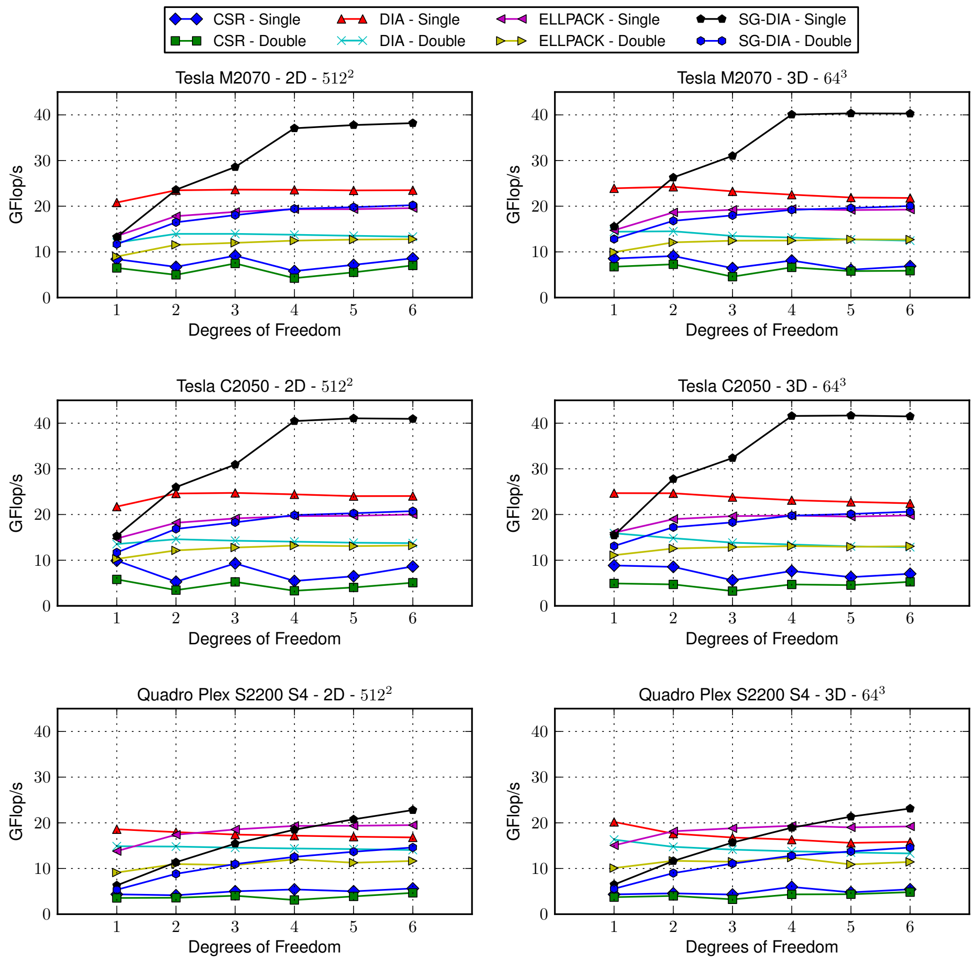


Fig. 4.1: Performance of SpMV using the new SG-DIA matrix format, compared with other sparse matrix formats in the Cusp library.

Table 4.2: Kernel specifications.

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

complete comparison of kernel performance for the kernels in Table 4.2, showing sim-

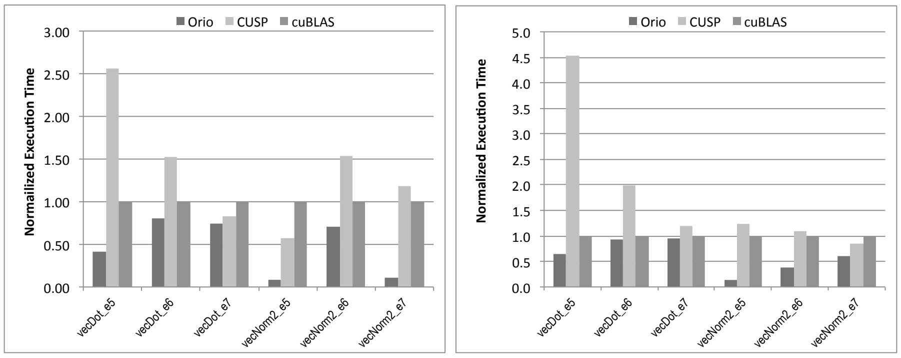


Fig. 4.2: Execution time for reduction kernels, normalized by the cuBLAS time (equal to 1.0 in these plots) on Tesla C2070 (left) and Tesla C1060 (right).

ilar improvements, is presented in [20].

4.2. Full Application Performance. We evaluate the performance of a full application solving a 3-D solid fuel ignition (SFI) problem, defined as follows.

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

A finite-difference approximation with a seven-point (star) stencil is used to discretize the boundary value problem in order to obtain a nonlinear system of equations. The system is then solved by using PETSc's Newton-Krylov iterative solvers.

The integration of the automatically generated and tuned CUDA implementations and corresponding host code is not fully automated at this time, but we plan to address it in the near future. For the complete application experiments, we integrated a subset of the kernels in Table 4.2 required for the linear system solution using restarted GMRES(m) with restart parameter $m = 30$, without preconditioning.[1] In addition to the linear solver kernels, in order to minimize CPU/GPU data transfers, we also generate optimized CUDA code for the two application-specific functions, `FormFunction` (stencil-based grid update) and `FormJacobian` (user-provided Jacobian computation), which are called by the nonlinear solver to update the sparse linear system and right-hand side. By automatically generating CUDA implementations of FormFunction and FormJacobian, we can keep all data on the GPU, thus avoiding the penalty of transfers at every nonlinear iteration.

Figure 4.3 compares the performance of the SFI application using two different GPU implementations: the version including autotuned kernel implementations (tuning parameters include threads per block, block count, and L1 size), based on the SG-DIA format described in Section 3, and the Cusp-based implementations included in the PETSc distribution. The results are normalized by the execution time for the CPU version compiled with Intel compilers (-O3 optimization level) and linked with the Intel MKL parallel library (using one thread per core on 8-core E5462 and E5430 Xeon machines). For small problem sizes, the transfer costs dominate, and a significant slowdown is observed in the GPU implementations. For larger problem

[1]The additional kernels required for preconditioning will be supported in the near future.

sizes (shown in Figure 4.3), the GPU versions outperform the CPU ones, with the autotuned version outperforming Cusp by up to a factor of 1.48 on the C2070 and up to a factor of 2.16 on the C1060.

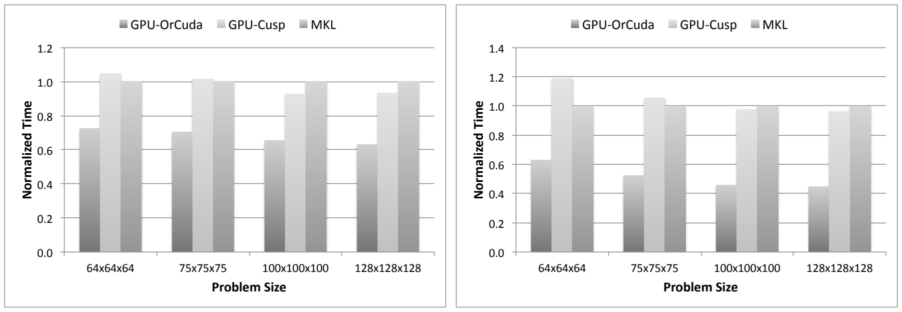


Fig. 4.3: Execution time for the SFI application, normalized by the CPU execution time (equal to 1.0 in these plots) on an 8-core E5462 Xeon/Tesla C2070 (left) and an 8-core Xeon E5430/Tesla C1060 (right).

5. Related Work.

Libraries such as Cusp [9,13], Thrust [10], and cuBLAS [23] 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. Our initial investigation [20] on tunable CUDA implementations achieved superior performance for several matrix and vector operations over these libraries. In our previous work [15], we demonstrated the performance scaling of our SG-DIA format across degrees of freedom and efficient bandwidth utilization on various GPU architectures.

The MAGMA project [21,26] aims to develop a library similar to LAPACK [3], 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.

Williams et al. [29] introduce an autotuning approach for optimizing the performance of sparse matrix-vector products on multicore processors, considering many parameters such as loop optimizations, SIMDization, and software prefetching. The results for a variety of matrix structures, including diagonal, show that architecture-specific optimizations consistently outperform off-the-shelf single approaches. Datta et al. [11] explore optimizations targeting various multicore processors, including GTX280 GPUs. The GPU optimizations and resulting performance gains at that time were limited by the GTX280 double-precision features and required CPU-GPU transfers.

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 [18,25]. 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.

Some commercial compilers and libraries have begun to exploit autotuning internally (e.g., Cray, Intel). However, that functionality is limited to vendor libraries and is not generally available to arbitrary codes. The significant exception among mainstream compilers is the open-source GNU Compiler Collection (GCC) [14] and specifically the Milepost GCC component [12], which employs a machine-learning-based approach that performs optimizations based on a set of code features. To our knowledge, GCC does not generate and optimize CUDA (or other GPU) code yet. The recent open-source release of the CUDA backend of nvcc presents interesting new opportunities for integrating application-specific autotuning.

6. Conclusions and Future Work. The work described here is the first necessary step toward enabling the semi-automated creation of libraries tuned for specific applications and target architectures. We show the performance benefit of employing autotuning in GPU code generation, with a focus on single functions with are computationally significant. The next steps are to increase the number of supportable kernels and extend Orio to enable tuning at higher levels of the call graph, enabling more types of optimizations, such as operation reordering and fusion of the underlying loops.

The autotuning process is not completely automatic. We are working on automating several of the currently manual steps, including the selection of kernel inputs to use in the tuning and the integration of the tuned code into the library and application source code (which requires relatively little effort but can be a barrier to wider use of autotuning).

Acknowledgments. This work was supported by the U.S. Department of Energy Office of Science under Contract No. DE-FOA-0000350 and DE-AC02-06CH11357. We thank Barry Smith of Argonne and other members of the PETSc team for fruitful discussions and ongoing support.

REFERENCES

[1] ELLPACK: Software for Solving Elliptic Problems. http://www.cs.purdue.edu/ellpack/, 2012. Last accessed June 30, 2012.
[2] Top 500 Supercomputing Sites. http://www.top500.org, 2012. Last accessed June 30, 2012.
[3] 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.
[4] P. Balaprakash, S. Wild, and B. Norris. SPAPT: Search problems in automatic performance tuning. In *Proceeding of the ICCS Workshop on Tools for Program Development and Analysis in Computational Science*, number Also available as Preprint ANL/MCS-P1872-0411, 2012.
[5] 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.
[6] 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.
[7] 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.
[8] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.
[9] N. Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2012. Version 0.3.0.
[10] 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.

[11] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[12] G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, and M. OBoyle. Milepost GCC: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39:296–327, 2011. 10.1007/s10766-010-0161-2.

[13] R. Galvez and G. van Anders. Accelerating the solution of families of shifted linear systems with CUDA. http://arxiv.org/abs/1102.2143, 2011.

[14] GNU Project. GCC, the GNU Compiler Collection. http://gcc.gnu.org/, 2012.

[15] 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.

[16] 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.

[17] Hypre. http://acts.nersc.gov/hypre/, 2012. Last accessed June 30, 2012.

[18] M. Khan, J. Chame, G. Rudy, C. Chen, M. Hall, and M. Hall. Automatic high-performance GPU code generation using CUDA-CHiLL, 2011. poster.

[19] 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.

[20] A. Mametianov, D. Lowell, C.-C. Ma, and B. Norris. Autotuning stencil-based computations on GPUs. In *Proceedings of IEEE International Conference on Cluster Computing (Cluster'12)*, 2012. To appear.

[21] 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.

[22] 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.

[23] NVIDIA. NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) Library. http://developer.nvidia.com/cublas, 2012. Last accessed April 28, 2012.

[24] NVIDIA. NVIDIA CUDA C Programming Guide Version 4.2, 2012.

[25] G. Rudy. *CUDA-CHiLL: A Programming Language Interface for GPGPU Optimizations and Code Generation*. PhD thesis, The University of Utah, Aug. 2010.

[26] 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.

[27] Trilinos. http://trilinos.sandia.gov/, 2012. Last accessed June 30, 2012.

[28] N. K. Vasileios Karakasis, Georgios Goumas. Performance models for blocked sparse matrix-vector multiplication kernels. In *Proceedings of the International Conference on Parallel Processing 2009*, 2009.

[29] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.