

# International Journal of Parallel Programming

## RT-CUDA: A Software Tool for CUDA Code Restructuring

--Manuscript Draft--

<b>Manuscript Number:</b>	IJPP-D-15-00004R2	
<b>Full Title:</b>	RT-CUDA: A Software Tool for CUDA Code Restructuring	
<b>Article Type:</b>	Original Research	
<b>Keywords:</b>	CUDA; GPGPU; NVIDIA Kepler; Massively Parallel Programming; Kernel Optimizations	
<b>Corresponding Author:</b>	Ayaz ul Hassan Khan, Ph.D. Qassim University Buraydah, Qassim SAUDI ARABIA	
<b>Corresponding Author Secondary Information:</b>		
<b>Corresponding Author's Institution:</b>	Qassim University	
<b>Corresponding Author's Secondary Institution:</b>		
<b>First Author:</b>	Ayaz ul Hassan Khan, Ph.D.	
<b>First Author Secondary Information:</b>		
<b>Order of Authors:</b>	Ayaz ul Hassan Khan, Ph.D.	
	Mayez Al-Mouhamed, PhD	
	Muhammed Al-Mulhem, PhD	
	Adel F. Ahmed, PhD	
<b>Order of Authors Secondary Information:</b>		
<b>Funding Information:</b>	King Abdulaziz City for Science and Technology (SA) (12-INF3008-04)	Prof. Mayez Al-Mouhamed
<b>Abstract:</b>	<p>Recent development in Graphic Processing Units (GPUs) has opened a new challenge in harnessing their computing power as a new general purpose computing paradigm. However, porting applications to CUDA remains a challenge to average programmers, which have to package code in separate functions, explicitly manage data transfers between the host and device memories, and manually optimize GPU memory utilization. In this paper, we propose a restructuring tool (RT-CUDA) that takes a C-like program and some user directives as compiler hints to produce an optimized CUDA code. The tool strategy is based on efficient management of the memory system to minimize data motion by managing the transfer between host and device, maximizing bandwidth for device memory accesses, and enhancing data locality and re-use of cached data using shared-memory and registers. Enhanced resource utilization is implemented by re-writing code as parametric kernels and use of efficient auto-tuning. The tool enables calling numerical libraries (CuBLAS, CuSPARSE, etc.) to help implement applications in science simulation like iterative linear algebra solvers. For the above applications, the tool implement an inter-block global synchronization which allow the execution overall among a few iterations which is helpful to balance load and to avoid polling. Evaluation of RT-CUDA has been performed using a variety of basic linear algebra operators (Madd, MM, MV, VV, etc.) as well as the programming of iterative solvers for systems of linear equations like Jacobi and Conjugate Gradient algorithms. Significant speedup has been achieved over other compilers like PGI OpenACC and GPGPU compilers for the above applications. Evaluation shows that generated kernels efficiently call math libraries and enable implementing complete iterative solvers. The tool help scientists developing parallel simulators like reservoir simulators, molecular dynamics, etc. without exposing to complexity of GPU and CUDA programming. We have partnership with a group of researchers at the Saudi Aramco, a national company in Saudi Arabia. RT-CUDA is currently explored as a potential</p>	

<b>Noname manuscript No.</b> (will be inserted by the editor)
--

# RT-CUDA: A Software Tool for CUDA Code Restructuring

Ayaz H. Khan<sup>a</sup> · Mayez Al-Mouhamed<sup>b</sup> ·  
Muhammed Al-Mulhem<sup>c</sup> · Adel F.  
Ahmed<sup>c</sup>

Received: date / Accepted: date

**Abstract** Recent development in Graphic Processing Units (GPUs) has opened a new challenge in harnessing their computing power as a new general purpose computing paradigm. However, porting applications to CUDA remains a challenge to average programmers, which have to package code in separate functions, explicitly manage data transfers between the host and device memories, and manually optimize GPU memory utilization. In this paper, we propose a restructuring tool (RT-CUDA) that takes a C-like program and some user directives as compiler hints to produce an optimized CUDA code. The tool strategy is based on efficient management of the memory system to minimize data motion by managing the transfer between host and device, maximizing bandwidth for device memory accesses, and enhancing data locality and re-use of cached data using shared-memory and registers. Enhanced resource utilization is implemented by re-writing code as parametric kernels and use of efficient auto-tuning. The tool enables calling numerical libraries (CuBLAS, CuSPARSE, etc.) to help implement applications in science simulation like iterative linear algebra solvers. For the above applications, the tool implement an inter-block global synchronization which allow the execution overall among a few iterations which is helpful to balance load and to avoid polling. Evaluation of RT-CUDA has been performed using a variety of basic linear algebra operators (Madd, MM, MV, VV, etc.) as well as the programming of iterative solvers for systems of linear equations like Jacobi and Conjugate Gradient algorithms. Significant speedup has been achieved over other compilers like PGI OpenACC and GPGPU compilers for the above applications. Eval-

<sup>a</sup>Computer Science Department

Qassim University

<sup>b</sup>Computer Engineering Department

<sup>c</sup>Information and Computer Science Department

King Fahd University of Petroleum and Minerals

Tel.: +966-13-8607633

Fax: +966-13-8602174

E-mail: ay.khan@qu.edu.sa, {mayez, mulhem, adelahmed}@kfupm.edu.sa

uation shows that generated kernels efficiently call math libraries and enable implementing complete iterative solvers. The tool help scientists developing parallel simulators like reservoir simulators, molecular dynamics, etc. without exposing to complexity of GPU and CUDA programming. We have partnership with a group of researchers at the Saudi Aramco, a national company in Saudi Arabia. RT-CUDA is currently explored as a potential development tool for applications involving linear algebra solvers by the above group. In addition, RT-CUDA is being used by Senior and Graduate students at King Fahd University of Petroleum and Minerals (KFUPM) in their projects as part of RT-CUDA continuous enhancement.

**Keywords** CUDA · GPGPU · nVidia Kepler · Massively Parallel Programming · Kernel Optimizations

## 1 Introduction

Recent development in Graphic Processing Units (GPUs) has opened a new challenge in harnessing their computing power as a new general purpose computing paradigm. Strong implications are expected on computational science and engineering, especially in the area of discrete numerical simulation.

Modern GPUs use multiple streaming multiprocessors (SMs) with potentially hundreds of cores, fast context switching, and high memory bandwidth to tolerate ever-increasing latencies to main memory by overlapping long-latency loads in stalled threads with useful computation in other threads. The Compute Unified Device Architecture (CUDA) is a simple C-like interface proposed for programming NVIDIA GPUs. However, porting applications to CUDA remains a challenge to average programmers. CUDA places on the programmer the burden of packaging GPU code in separate functions, of explicitly managing data transfer between the host and GPU memories, and of manually optimizing the utilization of the GPU memory.

There is a growing research for reducing the complexity of producing optimized CUDA kernels [5, 27, 35]. Even though the programming model of CUDA offer a more programmer friendly interface, programming GPUs is still considered error-prone and complex, in comparison to programming CPUs with standing parallel programming models, such as OpenMP [8, 34]. Most recently, quite a few directive-based GPU programming models have been proposed from both the research community (hiCUDA [15], OpenMPC [24], etc.) and industry (PGI Accelerator [36], HMPP [39], R-Stream [26], OpenACC, OpenMP for Accelerators [2], etc.). On the surface, these models appear to offer different levels of abstraction, and expected programming effort for code restructuring and optimization.

CUDA-lite [40] proposed a set of annotations to maximize the efficiency of the transformation. It takes as input a naïve CUDA code that treats the memory as a single entity instead of a hierarchical one. CUDA-lite performs the following transformations: inserts shared memory variables, loop tiling, memory coalesced loads and/or stores by replacing global memory access with cor-

responding shared memory access. hiCUDA [15] is a directive based language for programming NVIDIA GPUs that provides abstractions which are closely matched with the CUDA programming model. hiCUDA's goal is not to automate optimizations rather it makes it easier for the programmer to program CUDA. It still depends on explicit optimizations by the programmer, such as utilizing shared memory or constant memory. OpenMPC [24] proposed a set of directives to be considered along with OpenMP directives [2]. It optimizes data movement between CPU and GPU by applying the inter-procedural dataflow analysis to accomplish the following: 1) overriding transfers to GPU global memory when the global memory already have up to date values of relevant variables, and 2) overriding transfers to CPU from GPU memory that is written back results from GPU global memory to host memory if the relevant value wasn't used in the CPU part before it is written. It also performs parallel loop swap and loop collapsing to enhance the inter-thread locality. Furthermore, it also uses auto-tuning to obtain the final optimized CUDA code. PGI accelerator programming model [36] is a directive based model targeting general hardware accelerators, even though it currently supports only CUDA GPUs based on OpenACC standards. The PGI model allows very high-level abstraction similar to OpenMP. The user needs to insert directives into the host program to guide the compiler for particular set of kernel optimizations and code transformations. So, the user is required to be exposed to machine dependent GPU details such as kernel dimensionality, thread-block organization etc. OpenACC is the first standardization effort toward a directive-based, general accelerator programming model portable across device types and compiler vendors. HMPP [39] is another directive-based GPU programming model targeting both CUDA and OpenCL. It also provides very high-level abstraction on GPU programming similar to PGI accelerator. HMPP model is based on the concept of codelets, functions that can be remotely executed on hardware accelerators like GPUs. Because the codelet is a base unit containing computations to be offloaded to GPUs, porting existing applications using HMPP often requires manual modification of code structures, such as outlining of a code region or re-factoring existing functions. For optimized data management, HMPP uses the concept of a group of codelets. By grouping a set of codelets, data in the GPU memory can be shared among different codelets, without any additional data transfers between CPU and GPU. Combining the codelet grouping with HMPP advanced load/delegated store directives allows the same data-transfer optimizations as the data region in the PGI model. One unique feature in the HMPP model is that the same codelet can be optimized differently depending on its call sites. R-Stream [26] is a high-level, architecture-independent programming model that is based on the polyhedral model [3]. It targets various architectures, such as STI Cell, SMP with OpenMP directives, Tilera, and CUDA GPUs. R-Stream performs affine scheduling to transform the code into both fine-grained and coarse-grained parallelism. It performs following code optimizations: global memory coalescing, loop interchange, strip-mining, loop fusion, shared memory promotion and tiling. CUDA-CHiLL [23] is another compiler framework that performs transformations using a transfor-

Features		CUDA-lite	hiCUDA	OpenMPC	PGI	OpenACC	HMPP	R-Stream	CUDA-CHiLL	RT-CUDA
Code regions to be offloaded		None	Structured blocks	Structured blocks	Loops	Structured blocks	Loops	Loops	Loops	Structured blocks
Loop Mapping		None	Parallel	Parallel	Parallel Vector	Parallel Vector	Parallel	Parallel	Explicit	Parallel
Data Management	GPU memory allocation and free Data movement between CPU and GPU	None	Explicit	Explicit/Implicit	Explicit/Implicit	Explicit/Implicit	Explicit/Implicit	Implicit	Explicit	Implicit
Compiler Optimizations	Loop Transformations	Explicit	-	Explicit	Implicit	Imp-dep	Explicit	Implicit	Explicit	Explicit/Implicit
	Data Management Optimizations	Implicit	Implicit	Explicit/Implicit	Explicit/Implicit		Explicit/Implicit			Implicit
GPU-specific features	Thread Batching	Implicit	Explicit	Explicit/Implicit	Indirect/Implicit	Indirect/Implicit	Explicit/Implicit	Explicit/Implicit	Explicit	Implicit
	Utilization of special memories				Indirect/Implicit	Indirect/Imp-dep	Explicit	Implicit	None	Indirect/Explicit

Table 1: Summary of Features and Optimizations in Different Tools.

mation recipe that contains a set of commands for each required code transformation. Although it depends on command based transformations, applying these transformations still need some optimization heuristics [6] to be applied manually that constrain the optimization strategies considered and limit the possible values for the optimization parameters. Heuristics implemented in CUDA-CHiLL are Dependences and Parallelization, Global Memory Coalescing, Shared Memory and Bank Conflicts, and Maximize Reuse in Registers. It also suggests an auto-tuning framework to choose among resulting codes from various possibilities of parameters.

To understand the level of abstraction that each programming model/compiler provides, Table 1 summarizes the features and optimization approaches acquired in these compilers. Here, Explicit means that the feature is enabled by some user provided hints to the compiler, Implicit indicates that the compiler automatically handles the feature without user intervention, Indirect shows that the users can manually control the compiler to use the feature, and Imp-dep means that the feature is implementation dependent. The table shows that R-Stream provides the highest level of abstraction in comparison to other models/compilers found in literature as most of the features are handled implicitly in R-Stream. It also shows that hiCUDA and CUDA-CHiLL provide the lowest level of abstraction among other tools as the programmer has to control most of the features explicitly. However, lower level of abstraction may be beneficial in some cases that allow enough control over various optimizations and features specific to the underlying GPU architecture to achieve optimal performance. On the other hand, high level of abstraction sometimes limits the application coverage of the tool. RT-CUDA provides the same level of abstraction as R-Stream but also provides user-defined configurations to control various optimizations and features of the underlying GPU architecture to explore the effects of different kernel optimizations.

GPU Feature	Tesla K20c
Global Memory	4800 Mbytes
Total SMs	13
SP per SM	192
Total Cores	$192 \times 13 = 2496$
Shared Memory per Block	48 KB
Register per SM	65536
Warp Size	32
Max. Threads/Block	1024
Warps per SM	64
Max. Threads per SM	2048
Blocks per SM	16
L1 Cache	16 KB
L2 Cache	1280 KB

Table 2: GPU Specifications.

## 2 Background

### 2.1 GPU Architecture

Graphics Processing Units (GPUs) are organized into an array of highly threaded Streaming Multiprocessors (SMs). Each SM has a number of Streaming Processors (SPs) that share control logic and instruction cache. GPUs have their own memory hierarchy with unified memory request paths for loads and stores. This memory hierarchy has been developed from Level-0 (no hierarchy) to Level-3 (L1, L2, Global) with the advancement in the nVidia GPU generations (Tesla, Fermi, Kepler). Unlike CPU cache memory levels, GPU cache memories are read-only memories and doesn't have cache coherency protocols implemented. Table 2 shows the features of the GPU device (K20c based on nVidia Kepler Architecture) that we used in our experiments.

Like nVidia Fermi architecture, each SM in nVidia Kepler has 64 KB of on-chip memory that can be configured to be used as shared memory and L1 cache. Unlike nVidia Fermi, this memory can be configured as 32KB/32KB partitions for shared memory and L1 cache respectively in addition to 16KB/48KB and 48KB/16KB configurations [31]. L1 cache in nVidia Kepler can store only local memory accesses such as register spills and stack data, global memory loads are cached in L2 [1]. In addition to L2 cache, global loads can also be cached in compiler directed 48 KB read-only data cache. The read-only path can be managed automatically by the compiler or explicitly by the programmer.

The threads are grouped in blocks which are then scheduled to SMs dynamically on the availability of each SM. These threads follow the single-program multiple-data (SPMD) program execution model. Within a block, threads are grouped in 32-threads instruction called warps, where each warp is being executed in the single-instruction multiple-data (SIMD) manner. Each SP is responsible to execute one thread at a time. In nVidia Kepler, each SM contains four warp schedulers each with dual instruction dispatch units. So, each

SM can select four warps and two independent instructions per warp to be dispatched in each cycle.

## 2.2 CUDA Kernel Optimizations

CUDA kernel can be optimized in many possible ways with good understanding of the needs of the application. NVIDIA suggests a cyclic process namely APOD (Asses, Parallelize, Optimize, Deploy) [32] to help application developers to rapidly identify the portions of their code that would most readily benefit from GPU acceleration, rapidly realize that benefit, and begin leveraging the resulting speedups in production as early as possible. Using APOD, a programmer can apply and test the optimization strategies incrementally as they are learned. Optimizations can be applied at various levels, from overlapping data transfers with computation all the way down to fine-tuning floating-point operation sequences.

We have explored several optimizations that can be categorized into three classes based on their application:

1. **Manual:** This set of optimizations can only be applied by the programmer himself through manual code analysis and modifications. For example:
  - Vectorization can be applied by using vector data types provided in CUDA as structs and modify the code logic to use the correct element of the vector in other expressions.
  - Texture memory can be used in kernels by invoking texture intrinsic provided in CUDA such as `tex1D()`, `tex2D`, and `tex3D()` for 1D, 2D, and 3D arrays respectively. Also, it needs to be bound explicitly to CUDA array allocated in device memory.
  - To perform coalesced global memory, the programmer has to modify the algorithmic logic to change the access patterns of arrays. Although, it can also be achieved as a result of some other optimizations such as prefetching using shared memory which can be performed automatically by some tool provided hints from the user regarding shared memory size and array to be loaded.
  - To perform faster data exchange among threads within a warp, the programmer has to use nVidia Kepler's Shuffle Instructions explicitly in the code and change the code logic based on the data exchange requirements.
2. **User Driven:** This set of optimizations can be applied by an automatic process/compiler providing few hints from the programmer. For example:
  - Loop Collapsing is performed by merging double nested loops into a single loop which can be automatically done by loop analysis and transformation within the compiler.
  - Thread and Thread-Block Merging can be achieved by duplicating the global memory array expressions identified by the programmer with incrementing the array indices and modifying the main loop partitioning based on the thread granularity defined by the programmer.



Type	Optimization
Manual	Vectorization
	Texture Fetching
	Coalesced Global Memory Access
	nVidia Kepler Shuffle Instructions
User Driven	Loop Collapsing
	Thread / Thread-Block Merge
	Parallel Loop Swap
	Strip Mining
	Bank Conflict Free Shared Memory Access
Compiler	Using Read Only Data Cache
	Common Sub-Expression Elimination
	Loop Invariant Code Motion
	Loop Unrolling

Table 3: CUDA Kernel Optimizations Categorizations.

- Parallel Loop Swap can be automatically performed by loop analysis and transformations within the compiler.
- Strip Mining is a well known technique and can be applied automatically by the compiler using defined algorithms.
- Bank Conflict Free Shared Memory Access can be achieved by just padding one column in shared memory variable declarations.
- nVidia Kepler’s Read Only Data Cache can be utilized by using C99-standard “const \_\_restrict\_\_” keyword to any variable or data structure identified by the programmer.

3. **Compiler:** This set of optimizations is already applied within the current cuda compiler (nvcc) and applied automatically.

Table 3 categorizes these optimizations into different types. For further details of each of these optimizations, see Appendix A

### 3 RT-CUDA

In this section, we are presenting RT-CUDA in detail including its design specifications, restructuring transformations, and infrastructure.

#### 3.1 Design Specifications

To target the complex nature of the GPU architecture, programs often have to go through profound transformations. GPUs featured a massive number of active threads running in a machine that is primarily designed to process applications with abundant data parallelism. The existing compilers have mainly addressed the Basic Architectural (BA) optimizations of GPUs. The driving force behind the BA optimizations is to tailor the code to the uniqueness of the GPU architecture which favors the data parallel computing model. Most existing compilers have addressed five architectural features, which are:



Optimization Specifications	CUDA-lite	hiCUDA	OpenMPC	PGI	OpenACC	HMPP	R-Stream	CUDA-CHiLL	Usage Rate (%)	RT-CUDA
Input/Output GPU Memory Allocation	None	Medium	Medium	Medium	Medium	Medium	High	Low	58.33	High
Computation Partitioning and Decomposition	None	Medium	Medium	Medium	Medium	Medium	High	Low	58.33	High
Locality optimizations and Datacopy Transformations	High	Medium	Medium	Medium	Low	Medium	High	Low	66.67	High
Parallel Memory Bandwidth	High	High	High	High	High	High	High	High	100	High
Optimization of Architectural Parameters	None	None	None	Low	None	Low	Low	Medium	20.83	Medium
Use of automatic compiler optimization and/or programmer-guided optimization	Medium	High	Medium	Medium	High	Medium	Low	High	75	High
Synchronization across SMs	None	None	None	None	None	None	None	None	0	Medium
Invocation of Optimized external Libraries	None	None	None	None	None	None	None	None	0	High

Table 4: Optimization Specifications Abstraction Comparison.

- Data motion between the GPU accelerator and the host CPU (I/O).
- The thread organization as the block-level and thread-level parallelism and the mapping of threads to results (CP).
- A deep memory hierarchy ranging from a large register file, small shared memory, read-only memories, and a slow global memory (LO).
- A zero-overhead thread switching augmented with parallel memories and wide buses (PMB).
- User activation of some optimizations (CO/G): optimizations which are implemented using annotation-based language extensions, which enables the use of high-level languages while hiding to some extent the GPU details.

These optimizations are shown in top 5 rows of the Table 4, which shows the estimated optimization level (low, medium, and high) and percentage addressing each optimization (24/24=100%, 18/24= 75%, etc.). The overall usage rate of the optimization specifications is 47.4% (91/192). In the following, we summarize the BA optimizations that were implemented in most of the existing compilers:

1. **The Parallel Memory Bandwidth (PMB):** aims at mapping threads within a warp to access data in distinct storages in the device memory to enable coalesced global memory access. For shared memory accesses, map threads within a warp to access data in distinct memory banks to avoid serialization.
2. **The use of automatic compiler and/or programmer guided optimizations (CO/G):** it is becoming very attractive to provide user-based annotations for guide the compiler applying specific optimizations to a scope of code without much user exposure to details of the GPU hardware. Example of guided optimizations is the loop unrolling, nVidia Keplers data cache utilization, common-sub expression elimination, and etc.
3. **The locality optimization (LO):** aims at enhancing the use of the deep GPU memory hierarchy by (1) copying once data into shared memory to maximize data reuse while maintaining a data footprint that meets memory constraints, (2) apply blocking/tiling with a fixed maximum size to fit in

the cache, (3) efficient utilization of large register file, and (4) use special portions of GM that are the constant and texture memories.

4. **The Input/Output GPU Memory Allocation (I/O):** the use of inter-procedural data-flow analysis to optimize data movement between CPU and GPU, an explicit operation for many compilers. This includes allocating memory for GPU input and output and managing the explicit transfer of data between host CPU and device GPU.
5. **Computation Partitioning and Decomposition (CP):** (1) manage block-level and thread-level parallelism, (2) map block/kernel organization and dimension to results, (3) use of address transformations to map threads to results and adjust thread granularity to amortize transfer cost.

Although the BA optimizations are essential they are far from being sufficient to optimize simple domain specific applications. The target applications of our proposed code restructuring is the area of Iterative Linear Algebra Solvers (ILAS) for which the algorithms have the following features:

1. A spectrum of parallelism ranging from the abundance of data parallelism at the operator level (matrix multiplication (MM), Matrix vector (MV), inner product, vector scaling, etc.), tree like operations like sum of product, down to scalar execution.
2. A producer-consumer data layout in which data computed on in one iteration is mostly consumed in the same iteration with the possibility of forwarding to the next iteration for some other data.
3. Exact algorithm behavior requires enforcing some synchronization across all the thread blocks. However, more efficient execution can be made if iterations are allowed to overlap. This is especially useful when there is potential of load unbalancing, such as the case of sparse matrix vector operations.
4. The existence of highly optimized math libraries for basic dense and sparse linear algebra operations, which were developed by researchers and industry. Research based libraries are constantly enhanced to: cope with newer hardware features, improve data locality using operator code merging, and use low level programming in some critical code fragments.

To efficiently compile ILAS algorithms, we aim at a restructuring tool that embodies the BA optimizations and being able to address all the above domain specific (DS) optimizations. For this, we target integrating the above BA optimizations with the following additional compiler techniques:

1. **Optimization of Architectural Parameters (AP):** (1) carry out resource management and machine occupancy which is subject to a complex set of constraints, (2) determine the optimal values of kernel parameters using constraints analysis and/or auto-tuning.
2. **Inter-Block Synchronization (SYC):** since GPUs have no global synchronization, (1) use of customized inter-block synchronization when absolutely required for program correctness, (2) use of kernel entry/exit, lock-based, and lock free, and relaxed synchronization, (3) adapt synchronization to algorithm depending on expected degree of thread load unbalancing.

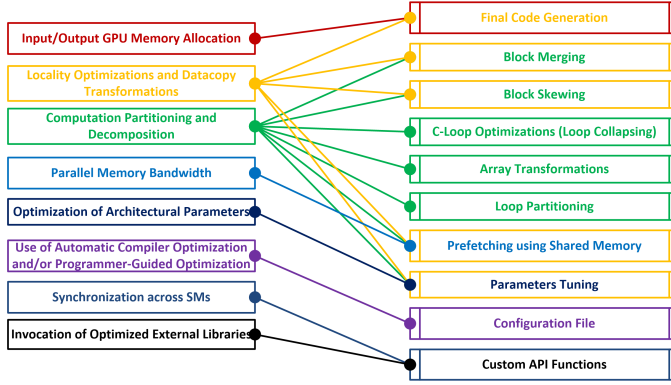


Fig. 1: RT-CUDA Code Transformation Strategy.

3. **Invocation of Optimized external Libraries (ELB):** some external libraries have been optimized at lower level programming and may deliver substantial performance advantages over manually optimized regular code. Library details (parameters and error handling) are hidden or abstracted to the user. Efficient invocation of external libraries require full understanding of its parameters and related implementation logic to select proper parameter values.

With these enhancements in the compiler design, RT-CUDA improves the overall usage rating of the optimization specifications from 47.4% to 52.32% (113/216) as shown in Table 4.

Figure 1 shows how all the above BA and DS optimizations can be integrated in the restructuring tool, see sections 3.2 and 3.3 for code transformation details.

Massively parallel GPUs have thousands of arithmetic units, capability to manage tens of thousands of hardware threads, and can achieve peak performance of several TFLOPS. Many highly computational workload having abundant data parallelism can be compiled into a large number of parallel threads to achieve high performance in terms of throughput on GPUs. Energy efficiency is becoming one key parameter to computing performance due to power scarcity in existing and future massively parallel systems [30, 46].

Reading or writing a value from GM require a factor of a hundredth fold more power than a floating point operation [12, 13, 20]. Moving a FP value across a chip costs a factor of five to ten than computing with it. As a result, keeping power consumption low will require minimizing data movement and enhancing temporal and spatial localities. One objective might be to minimize power consumption, which would likely require using the most appropriate type of core for the workload at hand. Another might be to minimize execution time. A third might be to balance both power consumption and execution time.

Our approach is based on enhancing program data manipulation by the compiler to improve spatial and temporal locality of data accesses in code re-

gions to improve power efficiency at runtime. Proposed restructuring technique aims at minimizing kernel execution time combined with the use of minimal data movement and enhanced temporal and spatial localities to improve power efficiency. RT-CUDA uses a set of six energy-aware rules (EORs), which are listed below together with their justifications:

1. **Minimizing data transfer overhead between CPU and GPU:** The CPU (host) and GPU (device) have separate memories. All data read/written on the device must be copied to/from the device (over the PCIe bus). This is very expensive, so it is important to try to minimize copies wherever possible, and keep data resident on device. This may involve porting more routines to device, even if they are not computationally expensive.
2. **Use of re-ordered synchronization to minimize the number of kernel invocations:** Kernel exit-re-entry (KEE) transfers control from the device back to CPU by interacting through the PCIe. KEE is sometimes a simple solution for inter-block synchronization, which is needed for a reduction or a collective write (CWs) to GM. In GPUs, data cached by a kernel cannot be used in a subsequent kernel launch because the TBs are dynamically assigned to SMs and data locality is lost at kernel exit. This leads to additional overhead for saving the data back into GM and reloading it again after kernel re-entry. A thread reference to some data requires a load instruction that reads GM and store into ShM, loads data into a register prior to use, and a store instruction to write back the register data into ShM. L1 cache is accessed while executing the load instruction. However, once a data is stored into ShM, L1 cache is no more accessed for the data. Instead of the KEE, we use an in-kernel approach using a custom inter-block synchronization, called re-ordered synchronization (ROS), to avoid the overhead associated with multiple KEEs and the loss of data locality. ROS enables staying in kernel without increasing the synchronization overheads as compared to KEE. Also, we may combine reductions and/or CWs to further reduce the number of needed ROS, if the algorithm allow.
3. **Load data from GM into ShM if there is sufficient data reuse, otherwise, data is loaded directly into Register File (RF):** Data load or store on GM or texture memory causes a 400-cycle long latency (LS) operation, while ALU operations, branch resolution, or accesses to ShM are short-latency (SL) events taking from 8 to 20 cycles. GPU put the burden on application programmers to explicitly manage the usage of shared and global memory. To use ShM we first must load data from GM to ShM, which is an LS. Next, every data invocation requires moving the data from ShM to the specific SP lane register, i.e. close to the execution unit. If there is no enough reuse of data that consumes more time and energy than simply loading data directly from GM into RF. Here, compiler stores short-lived working data in low energy structures such as RF, which is close to the execution units and recycle registers if data is not referenced any more. Data must be stored in RF just before its reference to avoid

power consumption due to long latency register context switching. Effectively leveraging ShM requires increasing kernel and thread grain size by combining operations, whenever possible. Short kernels and threads must have realistic memory usage using direct data load from GM into RF.

4. **Maximizing memory bandwidth using coalescing:** The memory bandwidth for the graphics memory on the GPU is high compared to the CPU, but there are many data-hungry cores so memory bandwidth is still a performance bottleneck. The maximum bandwidth is achieved when data is loaded for multiple threads in a single transaction, e.g. memory coalescing. This will happen when data access patterns meet certain conditions: 16 consecutive threads (i.e. a half-warp) must access data from within the same memory segment. This condition is met when consecutive threads read consecutive memory addresses within a warp. If the condition is not met, memory accesses are serialized, significantly degrading performance.
5. **Most frequently read-only data is kept in CM:** Constant Memory (CM) is a small read-only cache that caches a portion of GM. Maintaining some repeatedly accessed read-only (RO) data in CM helps to conserve memory bandwidth due to SRAM nature and its 20 cycles typical access time. Due to the small size of CM only RO data that has the highest read frequency should be saved in CM (and the rest in GM) before kernel entry.
6. **Auto-tuning to maximize occupancy and improve latency hiding:** use kernels that consists of arrays of thread blocks so to keep all cores in all SM busy. When programming for the GPU architecture, the programmer maps each loop iteration to a thread. Obviously, there must be at least as many total threads as cores, otherwise cores will be left idle. For best performance, the number of threads must be much greater than the number of cores. Accesses to global memory have several hundred cycles of latency when a thread stalls waiting for data, another thread can switch in this time to hide latency. NVIDIA GPUs feature very fast thread switching, and support many concurrent threads. The nVidia Fermi architecture supports up to 1536 concurrent threads on an SM, e.g. if there are 256 threads per block, it can run up to 6 blocks concurrently per SM (and the remaining blocks will queue). The resources must be shared between threads because high use of on-chip memory and registers will limit the number of concurrent threads. Typically, it is best to use many thousands of threads in total. The optimal number of threads per block should be found using auto-tuning which consists of running a parametric kernel for a few combination of architectural parameters (grid size, block size, thread grain size, and some compiler flags) and select the one that sustain the best performance.

The application of the above EORs in optimizing a few numerical algorithms like the SpMV, BiCG-Stab, and QMR is presented in the evaluation section 4.8. The energy consumption is compared to standard algorithm implementation using library calls with multiple-kernel invocations.

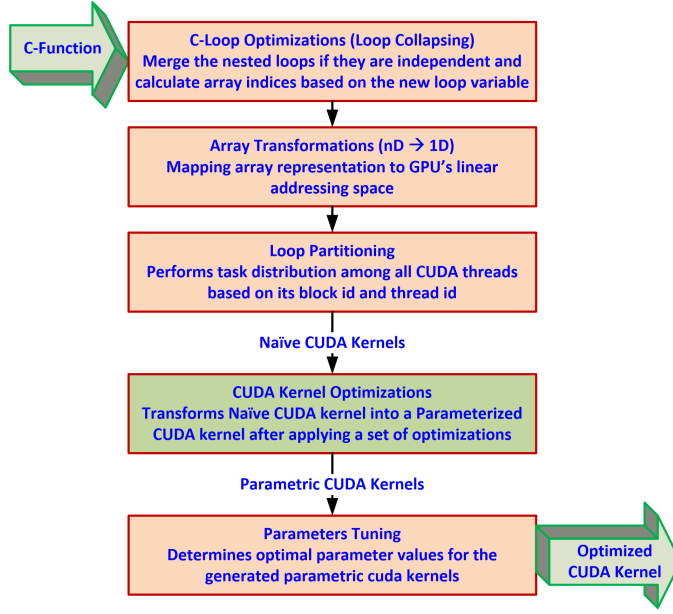


Fig. 2: Restructuring Tool Algorithm.

### 3.2 RTA-CUDA Description

RTA-CUDA (Restructuring Tool Algorithm for CUDA) has been developed to be implemented in a source-to-source transformation tool to convert a standard C function (input) containing computational loops into an optimized CUDA kernel (output) based on some user-defined variables. The algorithm consists of the following steps as shown in Fig. 2:

#### 3.2.1 C-Loop Optimizations (Loop Collapsing)

Merge the nested loops if they are independent and calculate array indices based on the new loop variable. For example, if  $i$  and  $j$  are two independent loops such that  $i = [0 \text{ to } N]$  and  $j = [0 \text{ to } N]$ . The new loop index ( $idx$ ) will be equal to  $[0 \text{ to } N * N]$  and  $i, j$  will be the quotient and remainder of the division of  $idx$  with  $N$  respectively such that ' $i$ ' represents row of the matrix and ' $j$ ' represents column of the matrix. So, instead of two nested loops, we will have now one main loop.

#### 3.2.2 Array Transformation ( $nD \rightarrow 1D$ )

In GPUs, device memory can be allocated only as linear memory so CUDA arrays are restricted to be allocated as 1D arrays while standard C language supports multi-dimensional arrays. In this step, all the multi-dimensional array

	bid = 0				bid = 1				bid = 2				bid = 3			
tid →	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
Loop Iterations →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Fig. 3: Task Distribution among all threads.

	bid = 0								bid = 1							
tid →	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
Loop Iterations →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Fig. 4: Task Distribution among all threads with block merging applied.

accesses in the expressions are converted to linear array representations. For example,  $C[i][j]$  will be represented as  $C[i * N + j]$  where  $N$  is the width of the array.

### 3.2.3 Loop Partitioning

Partition the main loop among all cuda threads. This obtains task distribution among all cuda threads based on its block id and thread id. Fig. 3 shows the task distribution among 4 cuda blocks with 4 threads per block. Each cuda thread identifies its working element of the resultant array using the block id and thread id within the cuda thread block. At this stage, each thread is mapped to one element of the resultant array. So, the loop is replaced by the statements to calculate the loop index to generate a Naïve CUDA Kernel.

### 3.2.4 CUDA Kernel Optimizations

In this step, each of the generated Naïve CUDA Kernel is transformed into a Parameterized CUDA Kernel after applying a set of optimizations as shown in Fig. 5.

#### 3.2.4.1 Block Merging

At this stage, each thread block is mapped to one block of resultant matrix/vector. Each thread within the block calculates one element of the resultant. To increase the thread granularity, each thread block can be mapped to multiple resultant blocks vertically. The number of blocks to be merged is defined as an input parameter and the optimal value of block merging can be obtained after running the parameter tuning algorithm as explained in section 3.2.5. Fig. 4 shows the task distribution among 2 cuda blocks with 4 threads per block after merging 2 resultant blocks that is each thread calculates two resultant elements at the same offset in consecutive resultant blocks. This is done by the following steps:

1. Convert the resultant variable into an array stored in local memory to compute the multiple elements simultaneously in a pipelined fashion.



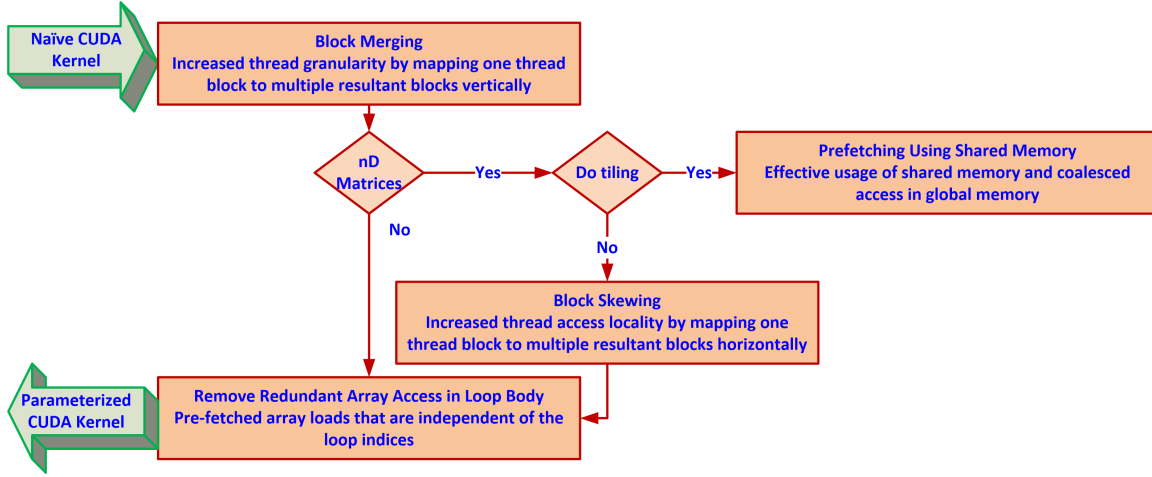


Fig. 5: CUDA Kernel Optimizations in RTA-CUDA.

2. Replace the first index of resultant matrix with the increment of loop index  $m$  where  $m$  defines the number of elements to be calculated by each thread.
3. Replace the resultant variable into array with loop index  $m$ .
4. Update row index calculations with multiple of number of blocks to be merged.

### 3.2.4.2 Prefetching using Shared Memory

To effectively use the shared memory and do coalesced access in global memory. The matrices in the loop that are going to be access by each thread in row-major can be tiled and loaded into shared memory with coalesced access. This is done by the following steps:

1. Declare shared variable for the tile of the given matrix to load the tiled rows of the number of blocks merged in the previous step.
2. Load the tile of the given matrix into shared memory by accessing the tiled rows in a coalesced manner such that each thread of the block access the consecutive elements in the same row.
3. Add barrier to synchronize all threads (`__syncthreads()`) after loading the tile.
4. Replace array access in the loop with the shared tile.
5. Add barrier to synchronize all threads (`__syncthreads()`) after calculating the tile.
6. Load the tile next consecutive tile of the given matrix into shared memory to be used in the computation of next iteration.
7. Add barrier to synchronize all threads (`__syncthreads()`) after calculating the tile.
8. Modify the main loop to traverse all tiles of the given matrix.

	bid = 0								bid = 1							
tid →	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3
Loop Iterations →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Fig. 6: Task Distribution among all threads with block skewing applied.

9. Calculate the remaining tile loaded in the last iteration.
10. unroll the loops to load and calculate the tile.

### 3.2.4.3 Block Skewing

To increase the thread access locality, each thread block can be mapped to multiple resultant blocks horizontally. The number of blocks to be skewed is defined as an input parameter and the optimal value of block skewing can be obtained after running the parameter tuning algorithm as explained in section 3.2.5. Fig. 6 shows the task distribution among 2 cuda blocks with 4 threads per block after skewing 2 resultant blocks that is each thread calculates two resultant elements at the consecutive offset in the resultant blocks. This is done by the following steps:

1. Convert the resultant variable into a matrix stored in local memory to hold the results of merged and skewed elements.
2. Replace the second index of resultant matrix with the increment of loop variable n where n defines the number of elements skewed in a resultant block.
3. Add second dimension of the resultant variable with loop index n.
4. Update column index calculations with multiple of number of blocks to be skewed.

### 3.2.4.4 Remove Redundant Array Access in Loop Body

At this stage, check for repeated load access of array elements within the newly created merged and skewed loops (m and n) in sections 3.2.4.1 and 3.2.4.3. If such an access is detected then replace it with local variable and move the loading of this element prior to the loop.

These steps generate a parameterized CUDA kernel with three parameters that are BLOCKSIZE (number of threads per block), MERGE\_LEVEL (number of blocks to be merged), and SKEW\_LEVEL (number of blocks to be skewed). The optimal values of these parameters can be obtained using the parameter tuning algorithm as explained in the following section 3.2.5.

### 3.2.5 Parameters Tuning Algorithm

Algorithm 1 determines the optimal parameters (BLOCKSIZE, MERGE\_LEVEL, and SKEW\_LEVEL) for the generated parametric CUDA kernel. The size of

cuda grid will be determined by dividing the total number of elements in the resultant array with the product of all three parameters.

The algorithm evaluates the generated parametric kernel with various possible combinations of BLOCKSIZE, MERGE\_LEVEL and SKEW\_LEVEL. The pruning of the list of possible parameters is used at three levels to reduce the repeated compilation and execution of the kernel. The three levels of pruning are as follows:

1. **Array Block Level:** This will skip those values of BLOCKSIZE which does not equally distribute the number of resultant elements among all threads (see step 3).
2. **Kernel Block Level:** This will skip those values of MERGE\_LEVEL and SKEW\_LEVEL which does not distributed the number of resultant elements among all kernel blocks (see step 9).
3. **Active Block Level:** This will skip those combinations of parameters which requires more than the available resources such as number of registers, shared memory and number of threads per SM (see step 31).

---

#### Algorithm 1 Parameters Tuning Algorithm

---

findOptimalParameters(N, CC)

---

##### Parameters:

N = Total Number of Elements in the Resultant Matrix

CC = Compute Capability of GPU Device

##### Constants and Keywords:

params = Structure of GPU Parameters

minBS = Minimum BLOCKSIZE, maxBS = Maximum BLOCKSIZE

minML = Minimum MERGE\_LEVEL, maxML = Maximum MERGE\_LEVEL

minSL = Minimum SKEW\_LEVEL, maxSL = Maximum SKEW\_LEVEL

KB = Kernel Blocks

RPT = Registers Per Thread

ShM = Shared Memory Per Block

RPB = Registers Per Block

WPB = Warps Per Block

ABW = Active Blocks Limit based on WPB

ABShM = Active Blocks Limit based on ShM

ABR = Active Blocks Limit based on RPB

CompleteParamsList = Structure Array for all Possible Kernel Parameters

CandidateParamsList = Structure Array for Candidate Kernel Parameters

OptimalParams = Structure for final Optimal Kernel Parameters

---

##### Algorithm:

```

1: Load params for compute capability of CC
2: for bs=minBS to maxBS Step *2 do
3:   if N mod bs ≠ 0 then
4:     continue
5:   end if
6:   for ml=minML to maxML Step *2 do
7:     for sl=minSL to maxSL Step *2 do
8:       KB = INT(N/bs/ml/sl)
9:       if KB = 0 then
10:        continue
11:       end if

```

---

---

```

12:      Compile the kernel only to determine the required RPT and ShM
13:       $RPB = \text{ROUND}(RPT \times bs, \text{params.RegisterAllocationUnitSize})$ 
14:       $WPB = \text{CEILING}(bs/\text{params.ThreadsPerWarp})$ 
15:       $ABW = \text{FLOOR}(\text{params.WarpsPerSM}/WPB)$ 
16:       $ABShM = \text{FLOOR}(\text{params.MaxSharedMemoryPerBlock}/ShM)$ 
17:       $ABR = \text{FLOOR}(\text{params.RegisterFileSize}/RPB)$ 
18:      Add  $\langle bs, ml, sl, ABW, ABShM, ABR \rangle$  into CompleteParamsList
19:    end for
20:  end for
21: end for
22: for all p in CompleteParamsList do
23:   if  $p.ABW > 0$  and  $p.ABShM > 0$  and  $p.ABR > 0$  then
24:     Add p into CandidateParamsList
25:   end if
26: end for
27: mintime = 0
28: for all p in CandidateParamsList do
29:   Execute the kernel with BLOCKSIZE=p.bs, MERGE_LEVEL=p.ml,
30:   SKEW_LEVEL=p.sl
31:   determine execution time (ktime) of the kernel
32:   if  $ktime > 0$  and ( $mintime = 0$  or  $mintime > ktime$ ) then
33:     mintime = ktime
34:     OptimalParams = p
35:   end if
36: end for

```

---

The algorithm takes kernel source file, number of resultant elements (N) and GPU Compute Capability (CC) for the target GPU device. It first loads the parameters for the given compute capability such as Register Allocation Unit Size, Threads Per Warp, Warps Per SM, Maximum Shared Memory Per Block, Register File Size, and etc (see step 1). It then loop over all possible combination of BLOCKSIZE, MERGE\_LEVEL, and SKEW\_LEVEL limiting to the range given by user with appropriate pruning (Array Block Level and Kernel Block Level) of the parameters as explained above. For each combination, it compiles the kernel with ptx information to determine the required number of Registers Per Thread (RPT) and Shared Memory (ShM) per block (see step 12). Then, calculate and store the restricted number of Active Blocks by Warp (ABW), Active Blocks by Shared Memory (ABShM), and Active Blocks by Registers (ABR) into a structured list (CompleteParamsList) (see steps 13 - 18). Then, it performs parameters pruning at Active Block Level and generate a list of possible optimal parameters (CandidateParamsList) (see steps 22 - 26). Finally, it execute the kernel for each combination of parameters in CandidateParamsList and determine the final optimal parameters (OptimalParams) that gives the minimum execution time (see steps 28 - 35).

### 3.3 RT-CUDA Design

RT-CUDA is a source-to-source transformation tool that is capable to convert a standard C-Program (input) into an Optimized CUDA Program (output).

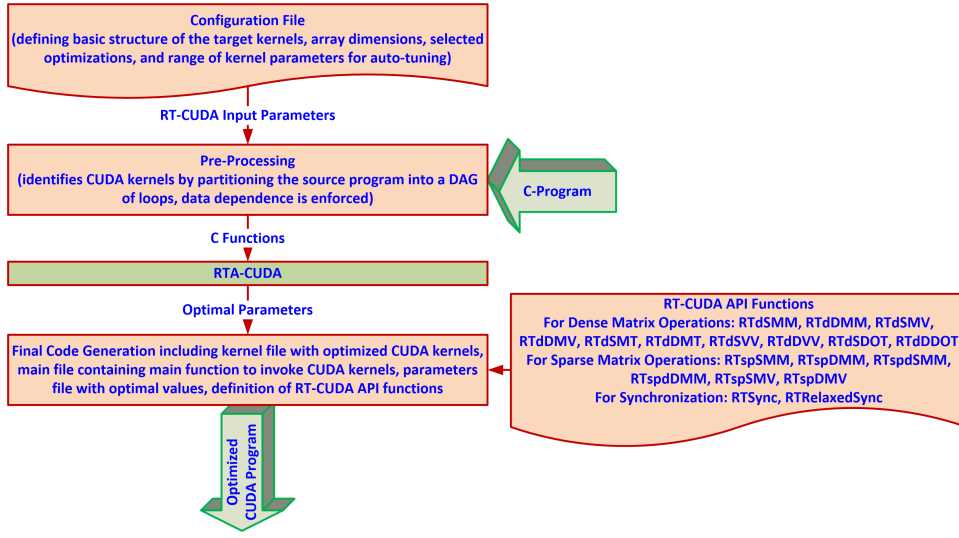


Fig. 7: Restructuring tool design.

The overall transformation is driven by some user-defined directives and API function calls provided in the tool with automatic kernel optimizations. The user is also able to include/exclude any of the optimizations available in the transformations. The tool follows the steps as shown in Fig. 7 to generate an Optimized CUDA Program.

1. **Configuration File:** A programmer defined text file containing configuration parameters to RT-CUDA for code transformations. This includes basic structure of the target kernel such as kernel name, semantics, used array dimensions and datatypes, selected optimizations, range of kernel parameters for auto-tuning, and target nVidia GPU architecture such as Fermi, Kepler, Maxwell, Tegra.
2. **Pre-Processing:** In this step, the source program is partitioned into a DAG (Dynamic Acyclic Graph) of loops identified as candidate CUDA kernels to be executed on GPU while separating the set of scalar segments that will be executed on host. Data dependence among the loops is enforced in the generated code. By examining the DAG, loops that are data independent can be merged together to reduce the exit/entry of kernels, copying data between Global Memory (GM) and Shared Memory (ShM), and to reduce loop overhead.
3. **RTA-CUDA:** This step will take each of the functions generated in the pre-processing step based on the loops identified as candidate CUDA kernels and apply RTA-CUDA algorithm as explained in section 3.2 to generate the optimized CUDA kernels.

API Function	Data Precision	Matrix Operation
RTdSMM	Single	Dense-Dense Matrix Multiplication
RTdDMM	Double	
RTdSMV	Single	Dense-Dense Matrix Vector Multiplication
RTdDMV	Double	
RTdSMT	Single	Dense Matrix Transposition
RTdDMT	Double	
RTdSVV	Single	Dense-Dense Vector Multiplication
RTdDVV	Double	
RTdSDOT	Single	Dot Product of Two Dense Vectors
RTdDDOT	Double	
RTspSMM	Single	Sparse-Sparse Matrix Multiplication
RTspDMM	Double	
RTspdSMM	Single	Sparse-Dense Matrix Multiplication
RTspdDMM	Double	
RTspdSMV	Single	Sparse-Dense Matrix Vector Multiplication
RTspdDMV	Double	

Table 5: Available API Functions in RT-CUDA for Dense and Sparse Matrix Operations.

4. **Final Code Generation:** At the end, generate the optimized CUDA program including all the optimized CUDA kernels obtained in RTA-CUDA step.
5. **RT-CUDA API Functions:** RT-CUDA also provides the functionality of transparent invocation of the most optimized external math library functions such CUBLAS and cuSparse for some of the dense and sparse matrix operations respectively. It provides interfacing APIs, error handling interpretation, and user transparent programming. These can be included by calling a related API function defined in the tool. Table 5 shows the list of available functions. It reduces the programming effort in terms of lines of code, error handling, and hides complex parameters selection by the programmers while giving similar performance in terms of execution time, see section 4.3 for code comparison and performance evaluations.

RT-CUDA also supports inter-block synchronization in three ways:

1. **CPU Synchronization:** This is the simplest approach recommended by Nvidia [16] for inter-block synchronization by exiting and re-entering the kernel that is considered as an implicit synchronization. This is done by defining separate CUDA kernels for each of the dependent loops and calling them in sequence from host.
2. **Lock-Free Synchronization:** This synchronization primitive is based on the work in [45]. It uses two arrays, named Ain and Aout, of length N for synchronization. When all threads of a block finish their work, the first thread of each block increments its location in the Ain array. Then, the first N threads of the first block in parallel check whether all blocks have written to their corresponding location in the Ain array. If so, these N threads write in parallel to the Aout array to inform other threads that the threads of this block have reached the synchronization point. Meanwhile,

the first thread of each block continuously checks its location in the Aout array until the value is set to k where k is the iteration number.

3. **Relaxed Synchronization:** We have developed a new synchronization primitive that can be useful in implementing iterative solvers with block dependencies among each iterations. Fig. 8 shows the flowchart of the relaxed synchronization. This approach overlaps the computation of two consecutive iterations. After completion of iteration 'k', each block start the computation of the iteration 'k+1' using the completed blocks of dependent array by the previous iteration. Each block updates its designated element in presence vector 'P' in global memory with the iteration number at the end of each iteration. So for the next iteration, it will call the relaxed synchronization primitive that will check the presence vector for the completed blocks of the previous iteration and return a work vector 'W' and the number of completed blocks 'bnum' that can be used to start the computation of the next iteration using the completed blocks of the previous iteration. The presence vector will be first loaded into shared memory from global memory with coalesced access to reduce global memory loads.

## 4 Performance Evaluation

We have run our experiments on Tesla K20c GPU (see Table-2 for specifications) with various applications including Demosaic, Histogram, Matrix Addition (Madd), Matrix Multiplication (MM), Matrix Vector Multiplication (MV), and Vector Vector Multiplication (VV). We have compared the implementations using RTA-CUDA with CUBLAS, GPGPU compiler, and OpenACC (PGI compiler) implementations. We have also evaluated the different inter-block synchronization primitives provided in the tool to be used in Jacobi Iterative Solver. Furthermore, the effects of calling external library functions for basic linear algebra operations and sparse matrices have also been evaluated. The correctness of the results of each converted application using RT-CUDA are guaranteed by comparing with the results of serial version of each application on CPU using the subset of the problem sizes. The result showed that our conversions produce correct resultant values. We have also trace the resultant matrix indices with the mapping of block and thread ids which are also found to be corrected.

### 4.1 Evaluation of Basic Linear Algebra Operations

This section shows the evaluation of the tool using a set of operators in LAPACK benchmark suite for basic linear algebra operations including Madd, MM, MV, and VV applications to compare with CUBLAS, GPGPU compiler, and OpenACC (PGI compiler). All four kernels are generated by the tool using RTA-CUDA algorithm as explained in section 3.2 and applied different set of transformations/optimizations according to the specific code structure of each



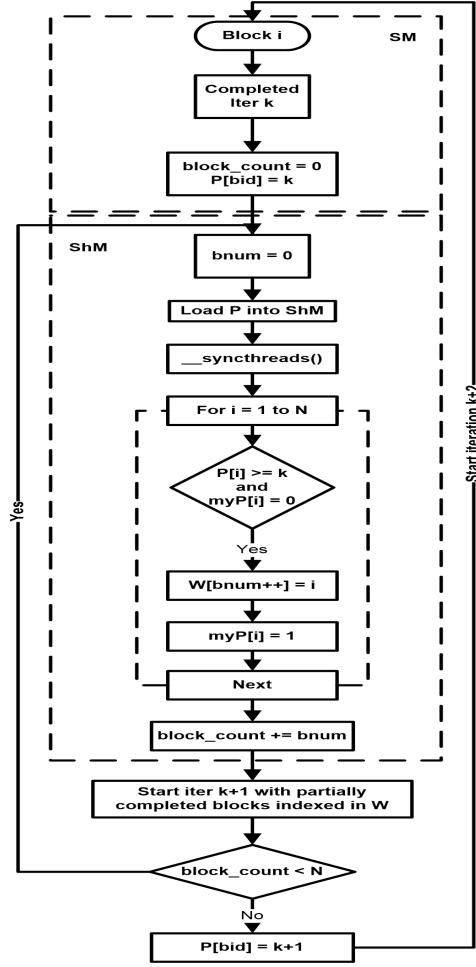


Fig. 8: Relaxed Synchronization.

application. Table 6 shows the applied transformations/optimizations for each application, the table also shows the optimal parameters ( $\langle \text{BLOCKSIZE}, \text{MERGE\_LEVEL}, \text{SKEW\_LEVEL} \rangle$ ) for each application obtained through parameter tuning algorithm (Algorithm 1). The comparisons for different applications and tools have been shown with appropriate space size (N) for each application to show the execution times in a particular range.

Fig. 9 shows the execution time in seconds of different applications using CUBLAS and RTA-CUDA. The comparisons have been performed using following CUBLAS functions:

- cublasSgeam for Madd
- cublasSgemm for MM and VV
- cublasSgemv for MV

Transformations/Optimizations	Madd	MM	MV	VV
Loop Collapsing	✓	✓		✓
Array Transformations	✓	✓	✓	✓
Loop Partitioning	✓	✓	✓	✓
Block Merging		✓		✓
Block Skewing	✓			
Prefetching using Shared Memory		✓	✓	
Remove Redundant Array Access		✓		✓
Read-Only Data Cache	✓	✓	✓	✓
Optimal Parameters	< 128, 1, 2 >	< 64, 16, 1 >	< 64, 1, 1 >	< 256, 32, 1 >

Table 6: Code Transformation Summary for Each Application.

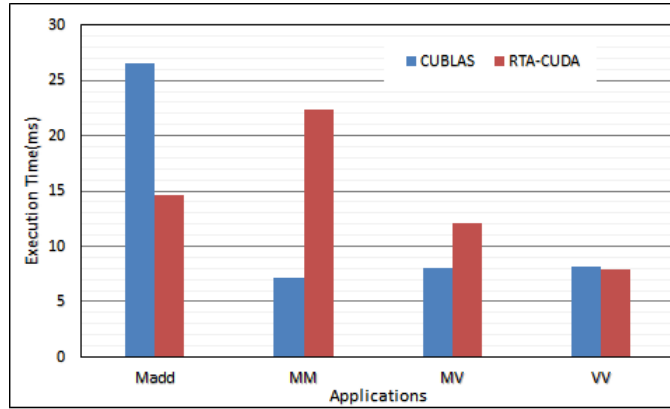


Fig. 9: Comparing CUBLAS and RTA-CUDA with different applications.

The results show that RTA-CUDA obtained better performance for Madd and VV. But, for complex applications such as MM and MV, CUBLAS still has significant performance advantage over RTA-CUDA. This is because cublasSgemm and cublasSgemv functions have been developed with complex kernel optimizations at very low level of coding by hand while at this stage, we are focusing on high level CUDA kernel optimizations. However, with the proposed high level kernel optimizations, RTA-CUDA outperforms CUBLAS with 45% improvement in case of Madd and 2% improvement in case of VV.

Fig. 10 shows the execution time in seconds of different applications using GPGPU compiler and RTA-CUDA. The results show that RTA-CUDA outperforms GPGPU compiler with 17% improvement in case of Demosaic, 30% improvement in case of MM, 99% improvement in case of MV and 50% improvement in case of VV. Also, MV implementation in GPGPU compiler gives value errors in case of large space size while RTA-CUDA generates correct values with any space size. So, the optimizations proposed in GPGPU compiler is not likely to be useful in nVidia Kepler family of GPUs where RTA-CUDA's optimizations obtained better performance.

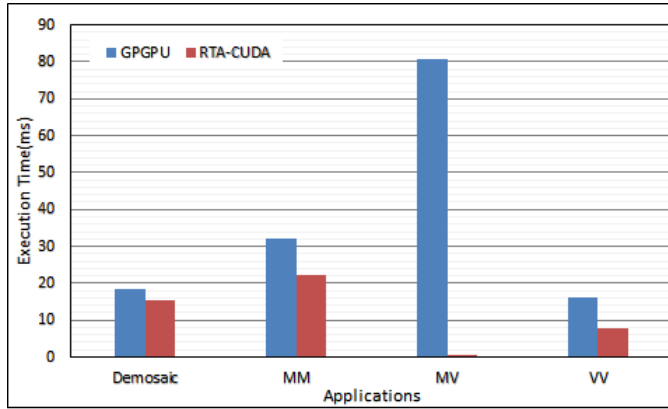


Fig. 10: Comparing GPGPU Compiler and RTA-CUDA with different applications.

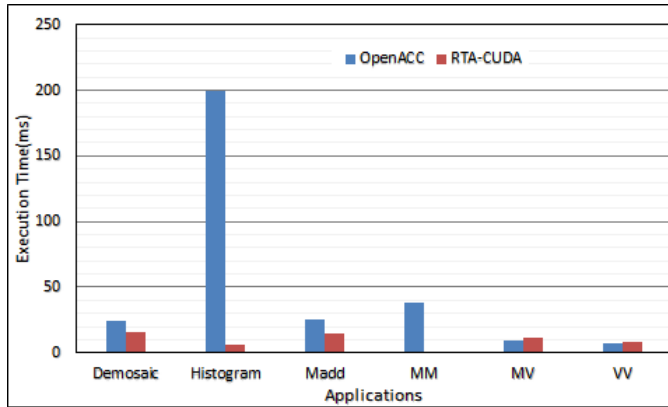


Fig. 11: Comparing OpenACC (PGI Compiler) and RTA-CUDA with different applications.

Fig. 11 shows the execution time in seconds of different applications using OpenACC implementation in PGI compiler and RTA-CUDA. The results show that RTA-CUDA outperforms OpenACC implementation of PGI compiler with 37% improvement in case of Demosaic, 97% improvement in case of Histogram, 42% improvement in case of Madd, and 99% improvement in case of MM and approx similar performance in case of VV and MV.

#### 4.2 Evaluation of Inter-Block Synchronization Primitives

This section shows the evaluation of inter-block synchronization primitives provided in the tool. The execution time of three variants of block Jacobi

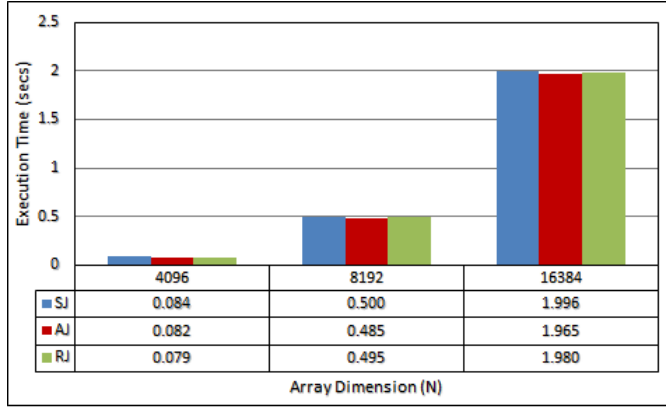


Fig. 12: Performance comparison of Single-Precision SJ, AJ, and RJ with different array dimensions.

iterative solver have been calculated and compared: 1) Synchronous Jacobi (SJ), 2) Asynchronous Jacobi (AJ), and 3) Relaxed Jacobi (RJ).

Fig. 12 shows the execution time in seconds of SJ, AJ, and RJ implementations with different array dimensions using 128 (maximum concurrent threads blocks possible for these implementations) number of blocks and single-precision floating point operations. Here, SJ implementation shows a little overhead of synchronization among each iteration and reduces with the increase in the array dimension that for  $N=16384$  with 128 thread blocks the synchronization overhead is just about 1.5% over AJ implementation. RJ implementation further obtained little improvement over SJ implementation that is about 1% reduction in execution time than SJ implementation except the case of  $N=4096$  where RJ improvement is about 6%. This shows that all thread blocks complete its execution with little difference in time as the tasks among each thread block is distributed equally. Relaxed synchronization approach is expected to give more performance improvement if the tasks are not evenly distributed among thread blocks on GPU architectures. To analyze the behaviour of RJ in case of unbalanced thread blocks execution, we have performed an experiment of a naïve block-row partitioning in sparse matrix-vector operation used in an iterative solver (such as Sparse Jacobi with MV), which causes some load unbalancing over the iteration space. Fig. 13 shows the execution time in seconds of SJ and RJ with unbalanced thread blocks. Here, RJ obtained about 8% performance improvement in terms of execution time over SJ implementation.

Fig. 14 shows the execution time in seconds of SJ, AJ, and RJ implementations with different array dimensions using 64 number of blocks (optimal number of thread blocks for these implementations) and double-precision floating point operations. Here, SJ implementation shows a high overhead of synchronization among each iteration that is for  $N=16384$  the synchronization overhead is about 12% over AJ implementation. RJ implementation is

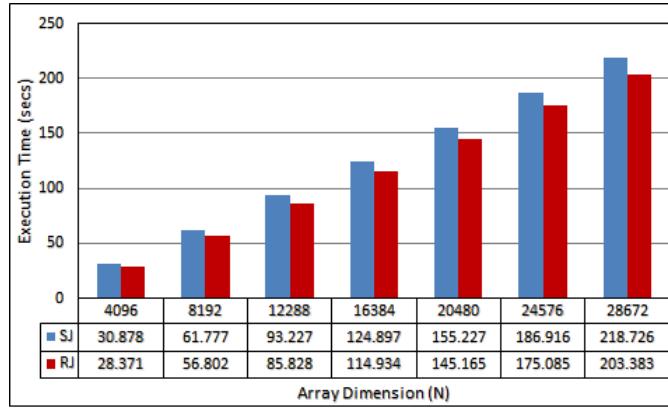


Fig. 13: Performance comparison of Unbalanced Blocks in SJ and RJ implementations.

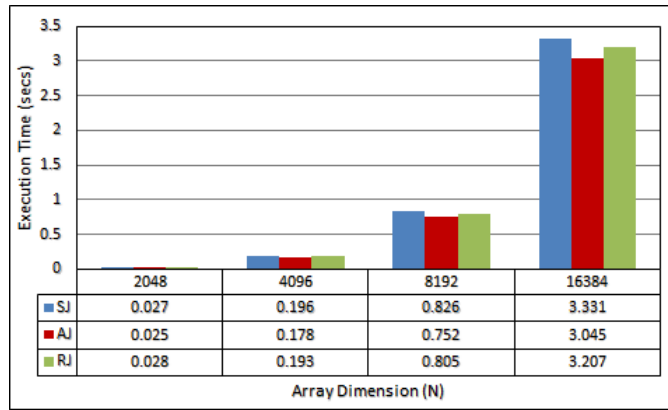


Fig. 14: Performance comparison of Double Precision SJ, AJ, and RJ with different array dimensions.

shown performance improvement of about 4% over SJ implementation. RJ also shows increasing trend in terms of performance improvement over SJ with the increase in the array dimension.

#### 4.3 Effects of Calling External CUBLAS Functions

As shown in section 4.1, in most of the cases CUBLAS library functions obtained the highest performance in comparison to RT-CUDA and other tools. This is because CUBLAS functions have been developed with complex kernel optimizations at very low level of coding by hand. To get the benefit of

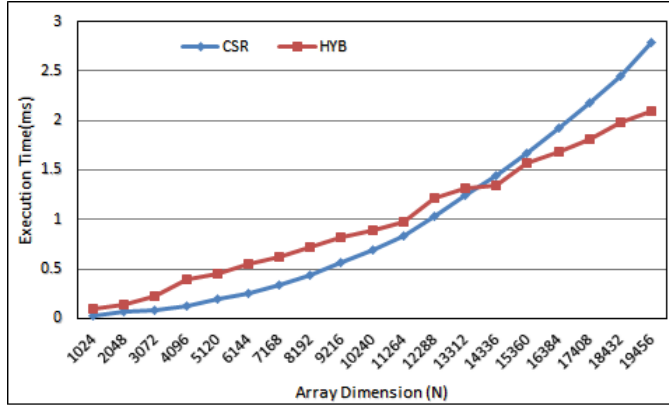


Fig. 15: Execution time in seconds of spdMV with CSR and HYB matrix formats.

the existing optimized CUDA libraries, we have provided a feature of calling external library functions as an optimization within the tool.

This section shows the performance evaluation of RT-CUDA using CUBLAS functions as an optimization instead of using RTA-CUDA algorithm. We applied the tool on Matrix Multiplication (MM), Matrix Vector Multiplication (MV), Matrix Transpose (MT), and Vector Vector Multiplication (VV). The execution time of each application has been compared with GPGPU compiler and PGI OpenACC compiler implementations.

RT-CUDA enables transparent invocation of the most optimized external math libraries like cuBLAS and cuSparse libraries. It provides interfacing APIs, error handling interpretation, and user transparent programming. An example of MV implementation in RT-CUDA (code available at [RT-CUDA MV Example](https://sites.google.com/site/ayazresearch/research)<sup>1</sup>) reduces the programming efforts of about 93% in terms of lines of code and hides complex parameters selection by the programmers while giving similar performance in terms of execution time. Fig. 16 shows normalized execution time of MM and MV operations using CUBLAS library and RT-CUDA API.

Table 9 shows the execution time in milliseconds of RT-CUDA, GPGPU compiler and PGI OpenACC implementations of different applications with different array dimensions. The results show that RT-CUDA significantly outperforms both GPGPU and PGI OpenACC implementations. It obtained performance improvements in terms of execution time of up to 80%, 100%, 4%, and 56% for MM, MV, MT, and VV respectively over GPGPU compiler with  $N=4096$ . It obtained performance improvements of up to 100%, 33%, 78%, and 70% for MM, MV, MT, and VV respectively over PGI OpenACC implementations with  $N=4096$ .

<sup>1</sup> <https://sites.google.com/site/ayazresearch/research>

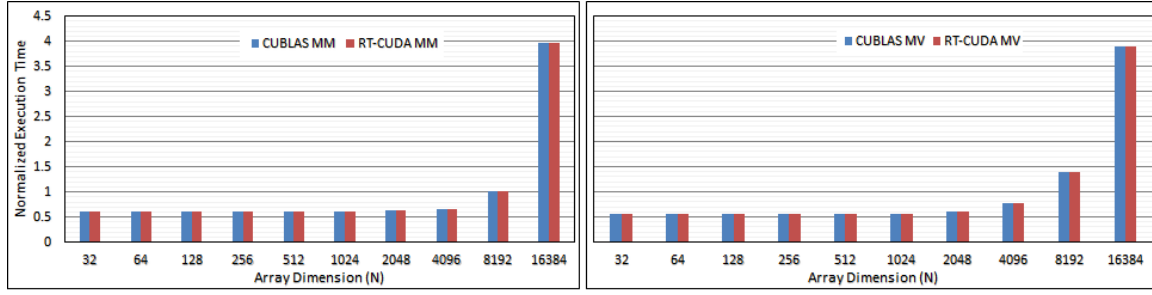


Fig. 16: Normalized Execution Time of MM (left) and MV (right) using cuBLAS library and RT-CUDA API.

N	Dense	CSR	BSR	HYB
1024	4	1	4	2
2048	16	4	16	6
4096	64	14	64	17
8192	256	51	257	54

Table 7: Matrix Storage Requirements in Different Formats with 90% Sparsity.

Sparsity	Single-Precision			Double-Precision		
	spMM	spdMM	spdMV	spMM	spdMM	spdMV
25%	382.7293	20.0087	0.0062	577.9461	33.6707	0.0072
50%	152.2969	13.9083	0.0042	230.9889	22.6241	0.0049
75%	36.6174	7.1978	0.0021	55.1729	11.4095	0.0023
90%	6.2449	3.0196	0.0009	9.0411	4.6379	0.0010

Table 8: Execution time in seconds for spMM, spdMM, and spdMV implemented in RT-CUDA for N=10240.

N	Matrix Multiplication			Matrix Vector Multiplication			Matrix Transpose			Vector Vector Multiplication		
	RT-CUDA	GPGPU	OpenACC	RT-CUDA	GPGPU	OpenACC	RT-CUDA	GPGPU	OpenACC	RT-CUDA	GPGPU	OpenACC
256	0.05	0.14	6.19	0.01	0.99	0.05	0.01	0.01	0.05	0.01	0.01	0.12
512	0.20	0.70	38.59	0.01	7.72	0.06	0.02	0.02	0.10	0.01	0.01	0.20
1024	1.10	3.80	542.95	0.04	80.93	0.10	0.07	0.06	0.30	0.03	0.07	0.37
2048	7.12	31.96	5361.62	0.14	659.42	0.21	0.25	0.25	1.11	0.11	0.26	0.71
4096	52.32	258.83	41468.49	0.51	5760.39	0.76	0.97	1.01	4.36	0.44	0.99	1.45

Table 9: Comparing RT-CUDA with GPGPU compiler and PGI OpenACC compiler using different applications.

#### 4.4 Effects of Sparse Matrix Operations using CUDA Sparse Library Routines

We have evaluated various sparse matrix formats available in cuSparse library for Sparse-Sparse Matrix Multiplication (spMM), Sparse-Dense Matrix Multi-



	Dense		Sparse					
			CSR			BSR		HYB
N	MM	MV	spMM	spdMM	spdMV	spdMM	spdMV	spdMV
1024	0.00117	0.00019	0.00865	0.00258	0.00013	0.01508	0.00156	0.00010
2048	0.00729	0.00026	0.04826	0.01831	0.00017	0.10783	0.00293	0.00014
4096	0.05251	0.00073	0.38818	0.17090	0.00013	0.84739	0.00622	0.00049
8192	0.41254	0.00176	3.17630	1.54086	0.00044	6.73315	0.01231	0.00072

Table 10: Execution time in seconds for different applications with available sparse formats in cuSparse library.

plication (spdMM), and Sparse-Dense Matrix Vector Multiplication (spdMV) (provided in RT-CUDA API). The objective of these evaluations is to access the performance of various matrix operators and storage schemes available in cuSparse library to select the best storages as standard in RT-CUDA. The evaluation results show that the sparse matrix multiplication (both spMM and spdMM) is only profitable in terms of memory allocations but not for execution time for computations as the dense matrix multiplication in CUBLAS is highly optimized and provide the best performance independent on the sparsity (percentage of number of zeros in the matrix) of the matrix. Whereas, in case of matrix-vector multiplication (spdMV), the sparse implementations obtain better performance both in terms of memory allocations and execution time in comparison to dense implementation.

Table 7 shows the memory allocations in MB for the sparse matrix in Dense, CSR, BSR (with 256 x 256 block dimensions), and HYB formats. Here, CSR and HYB formats show minimum memory requirements for matrix storage that is about 50-80% less than the dense and BSR formats. Table 10 shows the execution time in seconds for the computations of spMM, spdMM, and spdMV in Dense, CSR, BSR (with 256 x 256 block dimensions), and HYB formats. For matrix multiplication, sparse operations show significant overhead of computations due to irregular memory access patterns of the randomly initialized sparse matrices. For matrix-vector multiplication, sparse operations obtained the speedup of about 4 and 2.5 over dense operations for  $N=8192$  in CSR and HYB formats respectively. Furthermore, spdMV in CSR format is more efficient in terms of performance for  $N \leq 13312$  and spdMV in HYB format obtained more speedup for  $N \geq 14336$  as shown in Fig. 15.

#### 4.5 Generation of API Functions for Efficient Calling of cuSparse Library Routines

Since CSR and HYB sparse matrix formats have shown optimized storages (section 4.4), we decided to use them as implicit storage schemes in RT-CUDA. We have implemented API functions to call cuSparse library routines with CSR matrix format for spMM and spdMM operations, and with HYB matrix format for spdMV operation. Tables 8 show the execution time in seconds for






Matrix	Plot	Dimension	Non-Zeros	Sparsity
bcsstm13		2003	11973	99.70%
cavity10		2597	76367	98.87%
cavity17		4562	138187	99.34%
cdde1		961	4681	99.49%
coater1		1348	19457	98.93%

Table 11: Properties of the Sparse Matrices.

all three operations spMM, spdMM, and spdMV implemented in RT-CUDA with different matrix sparsity for  $N=10240$ .

Furthermore, RT-CUDA provides ability to load standard sparse matrices available in a matrix market file format (an ASCII-based file format designed to facilitate the exchange of matrix data) [38] into a sparse matrix structure to be used in RT-CUDA API functions for sparse matrix operations. We have evaluated the sparse operations of RT-CUDA using various standard sparse matrices in the domain of computational fluid dynamics available in the repository of University of Florida [9] and extracted from the real applications. Table 11 shows the properties of these matrices. All of the selected matrices has about 99% sparsity and are bend diagonal in nature. Fig. 17 shows the obtained speedup of RTspDMM, RTspdDMM, and RTspdDMV API functions of RT-CUDA (see Table 5 for details) over Dense equivalent operation. For RTspDMM, the sparse operation obtained more speedup if the non-zero elements are closed to diagonal as in the case of matrices cavity17 and cdde1 while in the case of bcsstm13, cavity10, and coater1 the speedup is relatively less due to scattered non-zero elements. For RTspdMM, the obtained speedup is seem to be dependent on the sparsity of the matrix. The matrices having more sparsity show more speedup than the matrices having less sparsity. For RTspdMV, the sparse operation obtained more speedup if the non-zero elements are closed to diagonal but with large dimension as in the case of cavity17 but the speedup drops significantly for small dimension as in the case of cdde1.

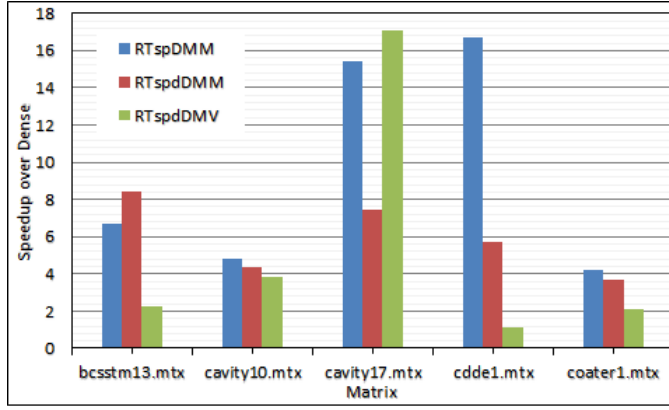


Fig. 17: Speedup of sparse MM and MV operations in RT-CUDA over Dense operations.

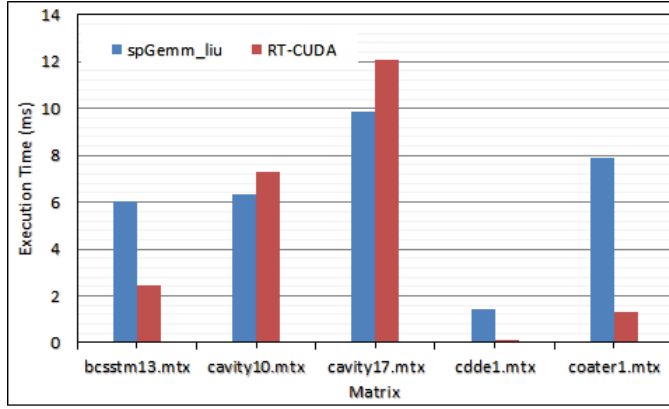


Fig. 18: Execution Time of MM using Standard Sparse Matrices.

We have also compared the sparse matrix multiplication implementation in RT-CUDA (RTspDMM) with an efficient sparse matrix multiplication recently proposed for x86-based many core processors [28] using a newly defined sparse storage format ESB (ELLPack Sparse Block format) with careful load balancing. Fig. 18 and Fig. 19 show the execution time in milliseconds of matrix multiplication implementation in RT-CUDA API and an efficient sparse GEMM implementation with ESB format using standard sparse matrices as shown in Table 11 and using hetpa diagonal sparse matrix with different dimensions respectively.

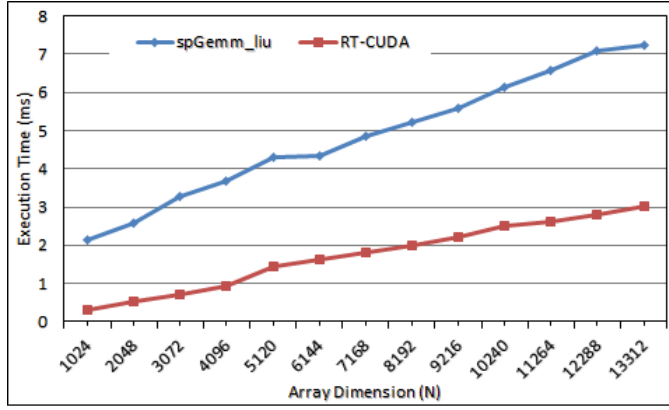


Fig. 19: Execution Time of MM using Hepta Sparse Matrix with different dimensions.

#### 4.6 Ease and Efficient Implementation of Large Scale Solvers

RT-CUDA hides architectural details of the underlying GPU device that helps traditional C programmers to develop parallel programs in a fast and efficient manner. RT-CUDA supports efficient development of sparse linear solvers such as conjugate gradient to be used in reservoir simulation softwares. It also includes API functions to allocate and initialize sparse matrices with random sparsity as well as reading matrix from matrix market file to be used as input for the solver. The implementation of such a solver can be optimized using the combination of user-defined functions and invoking highly optimized library functions including cuBLAS and cuSparse library functions. [RT-CUDA CG Implementation<sup>2</sup>](https://sites.google.com/site/ayazresearch/research) shows the inputs and outputs for the conjugate gradient algorithm implementation in RT-CUDA. The implementation uses the combination of user-defined functions for merged operations as an implementation optimization and RT-CUDA API functions for highly optimized sparse matrix-vector multiplication and dot product.

Fig. 20 shows the execution time in milliseconds for RT-CUDA Conjugate Gradient (CG) implementation in comparison to CG implementation in MAGMA library for hepta diagonal sparse matrix with different dimensions. The results show the hand-optimized MAGMA CG implementation provides 50% higher performance in terms of execution time than RT-CUDA implementation but increase in the array dimension to larger arrays reduces the performance gap between RT-CUDA and MAGMA implementations and provides same performance for large matrices to dimension  $N = 400384$ . This iterative CG algorithm is written in C and converted into optimized CUDA code with customized merged operations by using RT-CUDA which shows comparable performance to optimized MAGMA CG.

<sup>2</sup> <https://sites.google.com/site/ayazresearch/research>

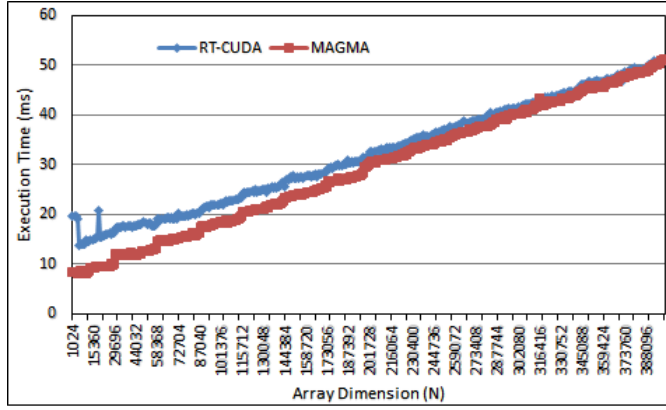


Fig. 20: Execution Time of Conjugate Gradient Implementation.

#### 4.7 Evaluation of the Tool Using More Elaborated Algorithms

##### 4.7.1 Hybrid Matrix Multiplication using Strassen and Winograd Recursive Algorithms

Using RT-CUDA, We implemented the hybrid matrix multiplication recursive algorithms based on Strassen (S-MM) and its Winograd (W-MM) variant. The above algorithms reduces the complexity of the canonical MM algorithm from  $O(N^3)$  to  $O(N^{2.8})$ . The implementation uses a depth first (DFS) traversal of a recursion tree where all cores work in parallel on computing each of the sub-matrices, which are computed in sequence. The DFS approach reduces the storage at the detriment of large data motion to gather and aggregate the results. RT-CUDA implementation uses a small set of basic algebra functions, invoking CuBLAS, and use of auto-tuning of parametric kernel to improve resource occupancy [22]. Evaluation show that RT-CUDA implementation of W-MM and S-MM with one recursion level outperform CUBLAS 5.5 library implementation with up to twice as fast for arrays satisfying  $N \geq 2048$  and  $N \geq 3072$  respectively. Based on the above it is clear that S-MM and W-MM RT-CUDA implementations with a few recursion levels can be used to further optimize the performance of basic algebra libraries. Compared to NVIDIA SDK library, S-MM and W-MM achieved a speedup between 20x to 80x for the above arrays.

Fig. 21 shows the speedup achieved by S-MM and W-MM over native CUBLAS Sgemm versus the matrix size using one recursion level. Note that W-MM shows the highest performance for the studied range of matrix size. CUBLAS is faster than our implementations only for matrices where  $N \leq 3072$  (for S-MM) and  $N \leq 2048$  (for W-MM). For matrices with  $N > 2048$ , W-MM becomes up to twice as fast as CUBLAS. S-MM becomes also faster (speedup of 1 to 1.9) than CUBLAS for  $N > 3072$ .

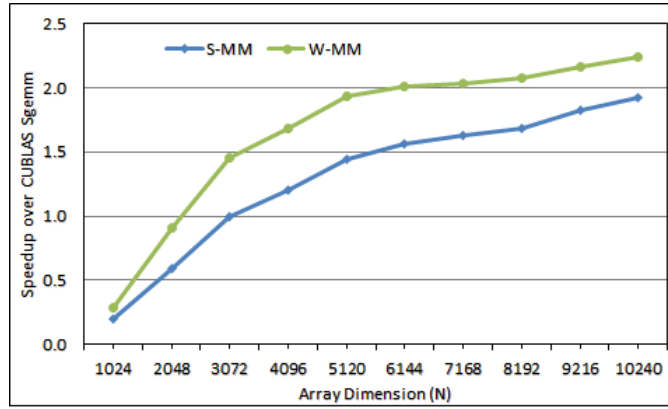


Fig. 21: Speedup of Hybrid Matrix Multiplication with 1-Level Recursion over CUBLAS Sgemm.

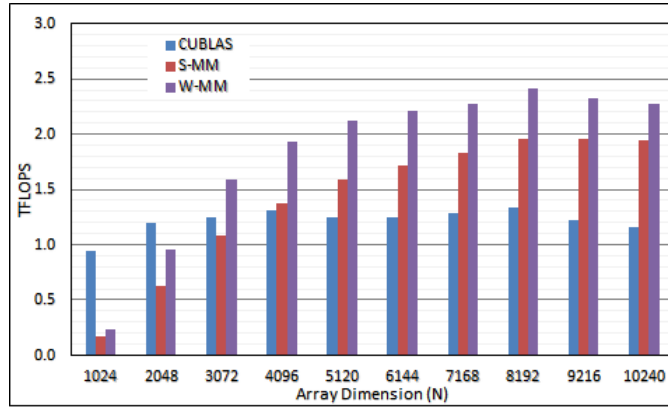


Fig. 22: Performance FLOPS for Hybrid Matrix Multiplication with 1-Level Recursion and CUBLAS.

Fig. 22 shows the performance in FLOPS for Strassen/Winograd implementation with CUBLAS versus the matrix size. Sorting the above implementations in descending order of best achieved performance gives W-MM, S-MM, and CUBLAS with best score 2.35 TFLOPS, 1.95 TFLOPS, and 1.35 TFLOPS respectively. The above performance scores represent 67%, 55%, and 38% of peak performance for nVidia Kepler K20c GPU (3.52 TFLOPS). These results confirm the effectiveness of proposed kernel optimizations because significant fraction of peak GPU performance is achieved.

Fig. 23 shows the speedup achieved by S-MM, W-MM, and CUBLAS Sgemm over NVIDIA SDK library MM(Matrix Multiplication) implementation. Notice that S-MM, W-MM, and CUBLAS are 80x, 60x, and 41x faster

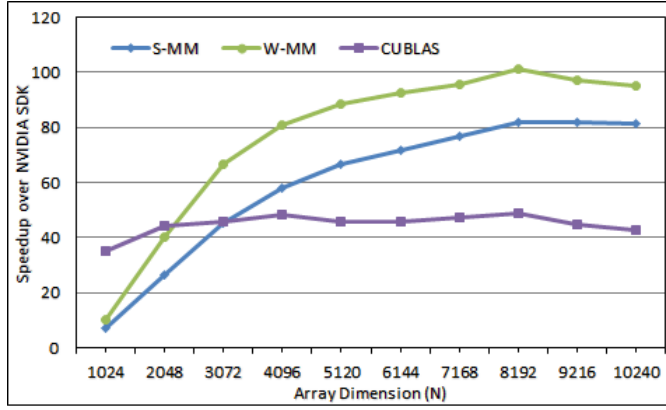


Fig. 23: Speedup of Hybrid Matrix Multiplication with 1-Level Recursion over NVIDIA SDK MM.

than NVIDIA SDK library when  $N \geq 4096$  respectively. Additionally, by inspecting Figs. 21 and 23 we note that the performance of our implementations scales much better than CUBLAS versus an increase in array size.

However, as we increase the level of recursion the execution time is also increased due to the overhead of additional operations in both S-MM and W-MM implementations. The reason is that the reduction in one matrix multiplication using Strassen and Winograd is not large enough to offset the overhead of the matrix additions and the stack overhead due to recursive invocations when two recursion levels or more are executed. Also note that the high-performance MM CUBLAS Sgemv is highly optimized at a very low level of programming.

#### 4.7.2 Matrix Transpose

Matrix Transpose is an important linear algebra function that has various applications in many numerical computing applications. Several factors hinder achieving optimal performance due to its inherent memory access patterns which cause bank conflicts in shared memory and prohibit the use of coalesced global memory access. RT-CUDA allowed to implement a few address permutation functions (APF) [21] that resolve the above limitations. Evaluation (Fig. 24) shows that RT-CUDA implementation of APF produces comparable performance to NVIDIA best implementation when using 32x32 tiles. The advantage of RT-CUDA implementation is the elimination of bank conflicts while allocating exactly the tile size memory space as opposed to best reported result which allocates for TxT tile an  $N \times (N+1)$  storage.

#### 4.7.3 Advanced Encryption Standard (AES-128)

Further to elaborate the generalization of RT-CUDA, we have converted the sequential implementation of the AES-128 ECB Encryption using RT-CUDA



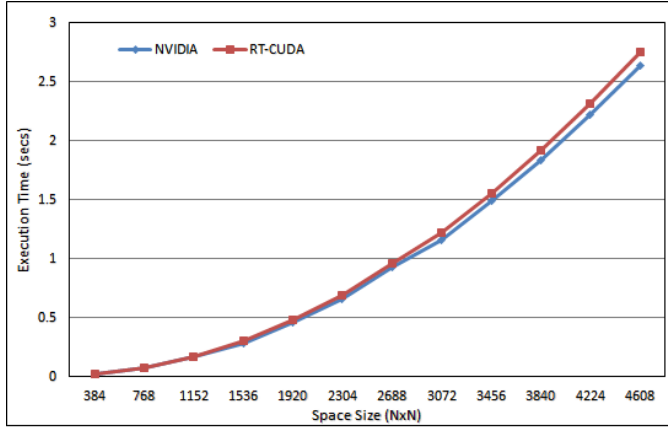


Fig. 24: Comparing RT-CUDA matrix transpose to best NVIDIA implementation.

and evaluated its performance on two of the recent GPUs (nVidia Fermi GPU: nVidia Quadro FX 7000 and nVidia Kepler GPU: Tesla K20c). We obtained a speedup of up to 87x against an advanced CPU (Intel Xeon X5690) based implementation. The AES algorithm basically comprises of four phases: (1) Key Expansion, (2) Initial Round, (3) Intermediate Rounds, and (4) Final Round. Readers are requested to [7] and [Rijndael-ammended.pdf from NIST] for details. In ECB mode, the forward cipher function (encryption) is done on each of the chunk or block of input plaintext separately. Since each block is processed independently of the others, this mode is exploitable for parallel implementation.

The AES 128-bit key is expanded in the host device (CPU) and stored in a 32-bit array of 44 elements, to be used as 128-bit round keys for 11 rounds. The host machine also divides plaintext data input into 16 bytes blocks; each contains 4 32-bit columns, which are all stored linearly in one dimensional array. The implementation loads expanded keys in constant memory while the t-tables (four 1KB lookup tables) in shared memory. The optimal number of blocks (Fermi=64, Kepler=1024) and threads (1024) are identified by RTA-CUDA.

Fig. 25 shows the speedup over CPU (sequential) implementation with both nVidia Fermi and nVidia Kepler GPUs using different input sizes. The RT-CUDA generated parallel implementations obtained speedup of up to 84 (on nVidia Fermi) and 87 (on nVidia Kepler) varies with different input sizes. However, for input data of low sizes (less than 1MB), the CPU and GPU performance is very similar.

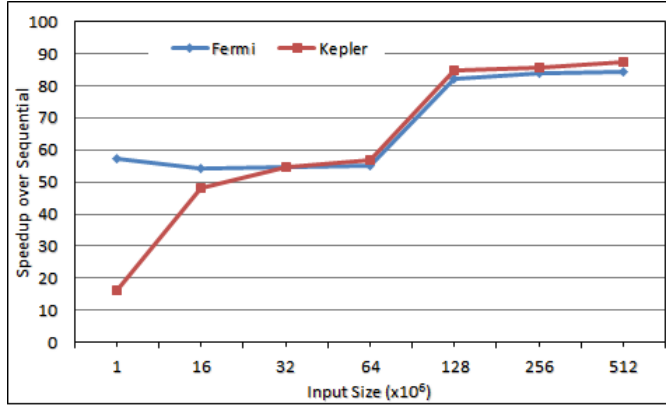


Fig. 25: AES-128 Speedup over CPU Implementation with both Fermi and Kepler GPUs.

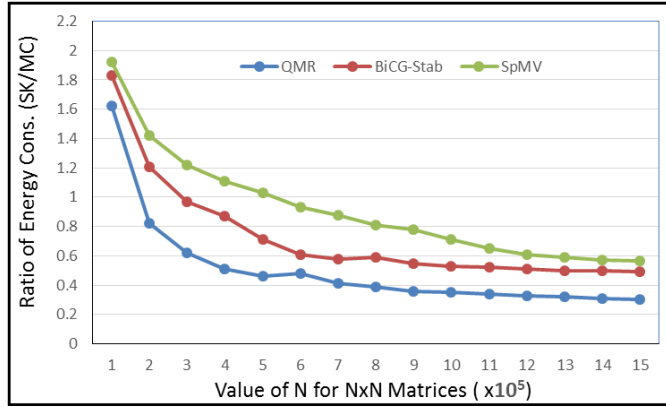


Fig. 26: Normalized Energy Consumption of various Algorithms.

#### 4.8 Energy Aware Code Optimization

Fig. 26 shows the normalized energy consumption for the SpMV, BiCG-Stab, and QMR algorithms. For each of the SpMV, BiCG-Stab and QMR algorithms, the plot shows the ratio of SK/MC, where SK and MC denotes the average energy consumption for the single-kernel implementation and multiple-kernel invocation using the CuSparse library. A reference sparse matrix is used with 10% non-zeros in each row. The Nvidia Management Library (NVML) and the utility (nvidia-smi) have been used to measure power and energy using a time function. MC uses a more optimized library operator coding than SK, but the implied data-CUDA implements SK using ROS global synchronization and in-kernel iteration to preserve the vector data locality across the iterations. Hence, SK is less optimized than CuSparse at the operator level, but its single-kernel

structure significantly enhances the memory access bandwidth, data locality, and data reuse.

Normally, a global synchronization (GS) is needed for each inner product (reduction) and after each collective write (CW) to GM. The application of EORs (section 3.1) seems to be effective for combining GSs to minimize overhead in addition to enhancing algorithm data locality. For example, the BiCG-Stab algorithm has five inner-products and two CW. Hence, there are seven kernel exit-entry in each iteration when BiCG is implemented using CuSparse. However, the application of EORs for optimizing BiCG-Stab energy consumption allowed implementing the main loop using only four GSs and preserving data locality due to in-kernel coding.

Proposed EOR memory management rules (section 3.1) produces the best results when the algorithm has more in-range vector operations which makes the best use of cached vector data. Each thread block updates many range vectors which are kept in ShM, thus saving the energy of repeatedly storing them back on GM before kernel exit and reloading at next kernel entry. Hence, SK implementation produces a saving of 58%, 41%, and 38% of energy consumption for QMR, BiCG-Stab, and SpMV respectively when the matrix size is large enough ( $N > 5 \times 10^5$ ) to achieve enough energy saving on GM accesses and to amortize the in-kernel synchronization overheads against a more optimized operator coding.

## 5 Conclusion

We have proposed a restructuring algorithm with best possible kernel optimizations and energy-aware rules. A design of a restructuring tool (RT-CUDA) has also been presented based on the proposed restructuring algorithm that is capable to convert a standard C-program (input) into an optimized CUDA program (output). Proposed restructuring technique aims at minimizing kernel execution time combined with the use of minimal data movement and enhanced temporal and spatial localities to improve power efficiency. The code transformations are driven by some user-defined configuration parameters and API function calls provided in the tool with automatic kernel optimizations. Evaluations have been performed using various numerical algebra kernels including matrix addition (Madd), matrix-matrix multiplication (MM), matrix-vector multiplication (MV), vector-vector multiplication (VV) and also two specialized kernels demosaic (Demo) and histogram (Hist). The generated optimized code gives significant improvement in performance in most of the cases. For Madd, we obtained 45% improvement over CUBLAS, and 42% over openACC. For VV, we obtained 2% improvement over CUBLAS, 50% over GPGPU compiler, and same performance as OpenACC. For Demo, we obtained 17% improvement over GPGPU compiler, and 37% over OpenACC. For MM, we obtained 30% improvement over GPGPU compiler, and 99% over openACC. For MV, we obtained 99% improvement over GPPGU compiler. For Hist, we obtained 97% improvement over OpenACC. RT-CUDA also pro-

vides the functionality of using BLAS routines of the highly optimized library CUBLAS. By using this feature, it obtained performance improvements in terms of execution time of up to 80%, 100%, 4%, and 56% for MM, MV, MT, and VV respectively over GPGPU compiler with N=4096. And it obtained performance improvements of up to 100%, 33%, 78%, and 70% for MM, MV, MT, and VV respectively over PGI OpenACC implementations with N=4096. The tool also supports sparse operations for Sparse-Sparse Matrix Multiplication (spMM), Sparse-Dense Matrix Multiplication (spdMM), and Sparse-Dense Matrix Vector Multiplication (spdMV) using the best possible sparse matrix formats available in cuSparse library. RT-CUDA enables transparent invocation of the most optimized external math libraries like cuBLAS and cuSparse. In this manner, RT-CUDA bridges the gap between optimized linear algebra libraries and high-level GPU programming. The code produced by RT-CUDA, in terms of readability, is much better than the code produced by other sophisticated compilation infrastructures for GPUs and it can be easily modified to apply complex optimizations specific to the underlying device architecture that bridges the performance gap between compiler and hand-optimized code. RT-CUDA facilitates the design of efficient parallel software for developing parallel simulators such as reservoir simulators, molecular dynamics, etc. RT-CUDA also has the capability to successfully generate GPU kernels for the Jetson TK1 (nVidia Tegra 4) which has 192 CUDA cores. Generated CUDA code is valid for all family on NVIDIA GPU including the Jetson TK1 mobile device. In future, we will evaluate RT-CUDA on other GPU architectures such as nVidia Maxwell and nVidia Tegra (Mobile GPUs) with some additional application classifications. Furthermore, we are intending to explore code transformations and optimizations for other vendors' architectures such as AMD GPUs that uses programming frameworks other than CUDA.

## A CUDA Kernel Optimizations

### A.1 Manual Optimizations

**Vectorization** improves the bandwidth utilization by using one of the vector data types such as float2, float4, float8 in CUDA as structs with some special data alignment [33, 44]. Global memory transactions in GPU are aligned to 128 bytes even if the actual data load is less than 128 bytes. So, if a specific thread within a warp performs 8 loads of float data type with sequence of load instructions then the GPU perform 8 different memory transactions. On the other hand, if a float8 type is used and perform single float8 load operation then it can be done by only one global memory transaction. So, to improve global memory bandwidth utilization, the programmer should use vector data types in the case when consecutive data loads are not aligned to 128 bytes. **Texture Fetching** utilized the texture memory that is a read-only portion of memory in device memory (DRAM) that has been cached (off-chip cache) on access [33, 44]. It has been accessible by all threads and host. It is opti-

mized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together achieve best performance. Texture references that are bound to CUDA arrays can be written to via surface-write operations by binding a surface to the same underlying CUDA array storage. Reading from a texture while writing to its underlying global memory array in the same kernel launch should be avoided because the texture caches are read-only and are not invalidated when the associated global memory is modified. So, texture fetches from addresses that have been written via global stores in the same kernel call returned undefined data. The data in texture can consist of 1, 2, or 4 elements of any of the following types: 1) Signed or unsigned 8, 16, or 32-bit integers, 2) 16-bit floating point values, and 3) 32-bit floating point values. Arrays declared in texture memory can be used in kernels by invoking texture intrinsic provided in CUDA such as `tex1D()`, `tex2D()`, and `tex3D()` for 1D, 2D, and 3D CUDA arrays respectively. Before invoking a kernel that uses texture memory, the texture must be bound to a CUDA array or device memory by calling `cudaBindTexture()`, `cudaBindTexture2D()`, or `cudaBindTextureToArray()`. **Coalesced Global Memory Access** refers to combining multiple memory accesses into a single transaction [14]. Global memory is the slowest memory on the GPU. Simultaneous global memory accesses by each thread of a half-warp (16 threads) during the execution of a single read and write instruction are coalesced into a single access. This is achieved based on the following conditions: 1) the size of the memory element accessed by each thread is either 4, 8, or 16 bytes, 2) the elements to be accessed form a contiguous block of memory, 3) the  $N^{th}$  element is accessed by the  $N^{th}$  thread in the half-warp, does not affect if any thread in between not accessing the global memory that is divergent warp, and 4) the address of the first element is aligned to 16 times the element's size. **nVidia Kepler's Shuffle Instructions** perform data exchange between threads within a warp [31]. It is more faster than the use of shared memory. This feature allows the threads of a warp to exchange data with each other directly without going through shared (or global) memory. So, this can present an attractive way for applications to rapidly interchange data among threads. There are four variants of shuffle instructions in CUDA that are `__shfl()`, `__shfl_up()`, `__shfl_down()`, and `__shfl_xor()`. Shuffle instructions can be used to free up shared memory to be used for other data or to increase warp occupancy and to perform warp-synchronous optimizations (removing `__syncthreads()`). All the `__shfl()` intrinsics take an optional width parameter which permits sub-division of the warp into segments. For example, to exchange data between 4 groups of 8 lanes in a SIMD manner. If width is less than 32 then each subsection of the warp behaves as a separate entity with a starting logical lane ID of 0. A thread may only exchange data with others in its own subsection. Width must have a value which is a power of 2 so that the warp can be subdivided equally; results are undefined if width is not a power of 2, or is a number greater than `warpSize`.

## A.2 User Driven Optimizations

**Loop Collapsing** is a technique to transform some nested loops into a single-nested loop to reduce loop overhead and improve runtime performance [19] specifically for irregular applications such as sparse matrix vector multiplication (spMV). Such applications pose challenges in achieving high performance on GPU programs because stream architectures are optimized for regular program patterns. It improves the performance of the application in three ways: 1) the amount of parallel work (the number of iterations, to be executed by GPU threads) is increased 2) inter-thread locality is increased 3) control flow divergence is eliminated, such that adjacent threads can be executed concurrently in an SIMD manner [25]. **Thread and Thread-Block Merging** enhance the data sharing among thread blocks to reduce the number of global memory accesses [47, 48]. Thread-Block Merge determines the workload for each thread block while Thread Merge decides the workload for each thread. If data sharing among neighbouring blocks is due to a global to shared memory (G2S) access, Thread-Block Merge should be preferred to better utilization of the shared memory. When data sharing is from a global to register (G2R) access, Thread Merge from neighbouring blocks should be preferred due to the reuse of registers. If there are many G2R accesses, which lead to data sharing among different thread blocks, the register file is not large enough to hold all of the reused data. In this case, Thread-Block Merge should be used and shared memory variables should be introduced to hold the shared data. In addition, if a block does not have enough threads, Thread-Block Merge instead of Thread Merge should also be used to increase the number of threads in a block even if there is no data sharing. Thread Merge achieves the effects of loop unrolling. It combines several threads' workload into one thread (combining  $N$  neighbouring blocks along column direction into one). By doing this, they can share not only shared memory but also the registers in the register file. Furthermore, some control flow statements and address computation can be reused, thereby further reducing the overall instruction count. The limitation is that an increased workload typically requires a higher number of registers, which may reduce the number of active threads that can fit in the hardware. **Parallel Loop Swap** is used to improve the performance of regular data accesses in nested loops [4, 25]. It uses to transform non-continuous memory accesses within the loop nest to a continuous memory access which is a candidate for the coalesced global memory access optimization. **Strip Mining** splits a loop into two nested loops [17, 41–43]. The outer loop has stride equal to the strip size and the inner loop has strides of the original loop within a strip. This technique is also used in loop tiling. In loop tiling or loop blocking, loops are also interchanged after performing strip mining to improve the locality of memory references that is why loop tiling is also called strip-mine-and-interchange. **Bank Conflict Free Shared Memory Access** improves performance by reordering the data into shared memory such that the memory addresses requested by the consecutive threads in a half-warp should be mapped to different memory banks of shared memory [11]. Shared

memory banks are organized such that successive 32-bit words are assigned to successive banks and the bandwidth is 32 bits per bank per clock cycle. In GPUs, the warp size is 32 threads and the number of banks is 16. So, a shared memory request for a warp is split into one request for the first half of the warp and one request for the second half of the warp. However, no bank conflict occurs if only one memory location per bank is accessed by a half warp of threads. **Using Read-Only Data Cache**, introduced in nVidia Kepler in addition to L1 cache, can benefit the performance of bandwidth-limited kernels [1,31]. This is the same cache used by the texture pipeline via a standard pointer without the need to bind a texture beforehand and without the sizing limitations of standard textures. This feature is automatically enabled and managed by the compiler, access to any variable or data structure that is known to be constant through programmer use of the C99-standard “const \_\_restrict\_\_” keyword tagged by the compiler to be loaded through constant data cache.

### A.3 Compiler Optimizations

**Common Sub-Expression Elimination** is a compiler optimization technique that searches for instances of identical expressions, evaluates to the same value, and replace them with a single variable holding the computed value [10,49]. It enhances the application performance by reducing the number of floating point operations. In CUDA, common sub-expression elimination can be used to avoid redundant calculations for the initial address of an array. **Loop Invariant Code Motion** (also called hoisting or scalar promotion) is a compiler optimization that has been performed automatically [18,37]. Loop invariant code is a set of statements or expressions within the body of a loop that can be moved outside of the body without affecting the semantics of the program. It makes loops faster by reducing the amount of code that executes in each iteration of the loop. The CUDA C compiler automatically applies this optimization technique to the PTX code. **Loop Unrolling** is a compiler optimization technique that is applied for the known trip counts at the compile time either by using the constants or templating the kernel [29]. NVIDIA compiler also provides a directive ‘#pragma unroll’ to explicitly activate the loop unrolling on a particular loop.

### Acknowledgment

The authors would like to acknowledge the support provided by King Abdulaziz City for Science and Technology (KACST) through the Science & Technology Unit at King Fahd University of Petroleum & Minerals (KFUPM) for funding this work through project No.12-INF3008-04 as part of the National Science, Technology and Innovation Plan. We are also very thankful to Mr. Anas Al-Mousa for providing the code implementations in OpenACC and also



thankful to King Abdullah University of Science and Technology (KAUST) for providing access to their K20X GPU cluster to run the experiments.

## References

1. Tuning cuda applications for kepler. <http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html>. Accessed: 10-06-2013
2. Beyer, J.C., Stotzer, E.J., Hart, A., de Supinski, B.R.: Openmp for accelerators. In: IWOMP, Lecture Notes in Computer Science, pp. 108–121. Springer (2011)
3. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.* **43**(6), 101–113 (2008). DOI 10.1145/1379022.1375595
4. van den Braak, G., Mesman, B., Corporaal, H.: Compile-time gpu memory access optimizations. In: Embedded Computer Systems (SAMOS), 2010 International Conference on, pp. 200–207 (2010). DOI 10.1109/ICSAMOS.2010.5642066
5. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for gpus: Stream computing on graphics hardware. *ACM Trans. Graph.* **23**(3), 777–786 (2004). DOI 10.1145/1015706.1015800
6. C. Chen, J.C., Hall, M.: Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In: International Symposium on Code Generation and Optimization (2005)
7. Daemen, J., Rijmen, V.: The design of Rijndael: AES — the Advanced Encryption Standard. Springer-Verlag (2002)
8. Dagum, L., Menon, R.: Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.* **5**(1), 46–55 (1998). DOI 10.1109/99.660313
9. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. *ACM Trans. Math. Softw.* **38**(1), 1:1–1:25 (2011). DOI 10.1145/2049662.2049663. URL <http://doi.acm.org/10.1145/2049662.2049663>
10. Ershov, A.P.: On programming of arithmetic operations. *Commun. ACM* **1**(8), 3–6 (1958). DOI 10.1145/368892.368907
11. Farivar, R., Campbell, R.: Plasma: Shared memory dynamic allocation and bank-conflict-free access in gpus. In: Parallel Processing Workshops (ICPPW), 2012 41st International Conference on, pp. 612–613 (2012). DOI 10.1109/ICPPW.2012.94
12. Gebhart, M., Johnson, D.R., Tarjan, D., Keckler, S.W., Dally, W.J., Lindholm, E., Skadron, K.: A hierarchical thread scheduler and register file for energy-efficient throughput processors. *ACM Trans. Comput. Syst.* **30**(2), 8:1–8:38 (2012). DOI 10.1145/2166879.2166882
13. Gray, A., Sjostrom, A., Llieva-Litova, N.: Best practice mini-guide accelerated clusters: Using general purpose gpus. Tech. rep., University of Warsaw (2013)
14. Ha, P.H., Tsigas, P., Anshus, O.J.: The synchronization power of coalesced memory accesses. In: Taubenfeld, G. (ed.) *DISC, Lecture Notes in Computer Science*, vol. 5218, pp. 320–334. Springer (2008)
15. Han, T.D., Abdelrahman, T.S.: hicuda: A high-level directive-based language for gpu programming. In: Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, pp. 52–61. ACM, New York, NY, USA (2009). DOI 10.1145/1513895.1513902
16. Harris, M.: Optimizing Parallel Reduction in CUDA. Tech. rep., nVidia (2008). URL <http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>
17. Hennessy, J., Patterson, D., Asanović, K.: Computer Architecture: A Quantitative Approach. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann/Elsevier (2012)
18. Ikeda, T., Ino, F., Hagihara, K.: A code motion technique for accelerating general-purpose computation on the gpu. In: In Proceedings of the International Parallel and Distributed Processing Symposium, pp. 1–10 (2006)
19. Jackson, A., Agathokleous, O.: Dynamic loop parallelisation. *CoRR* **abs/1205.2367** (2012)



20. Kasichayanula, K., Terpstra, D., Luszczek, P., Tomov, S., Moore, S., Peterson, G.: Power aware computing on gpus. In: Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on, pp. 64–73 (2012). DOI 10.1109/SAAHPC.2012.26
21. Khan, A., Al-Mouhamed, M., Fatayar, A., Almousa, A., Baqais, A., Assayony, M.: Padding free bank conflict resolution for cuda-based matrix transpose algorithm. In: Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014 15th IEEE/ACIS International Conference on, pp. 1–6 (2014). DOI 10.1109/SNPD.2014.6888709
22. Khan, A.H., Al-Mouhamed, M., Fatayer, A., Mohammad, N.: Optimizing the matrix multiply using strassen and winograd algorithms with limited recursions on many core. to appear in International Journal of Parallel Programming (IJPP) (2015)
23. Khan, M., Basu, P., Rudy, G., Hall, M., Chen, C., Chame, J.: A script-based autotuning compiler system to generate high-performance cuda code. ACM Trans. Archit. Code Optim. **9**(4), 31:1–31:25 (2013). DOI 10.1145/2400682.2400690
24. Lee, S., Eigenmann, R.: Openmpc: extended openmp for efficient programming and tuning on gpus. International Journal of Computer Science and Engineering (IJCSE) **8**(1), 4–20 (2013)
25. Lee, S., Min, S.J., Eigenmann, R.: Openmp to gpgpu: A compiler framework for automatic translation and optimization. SIGPLAN Not. **44**(4), 101–110 (2009). DOI 10.1145/1594835.1504194
26. Leung, A., Vasilache, N., Meister, B., Baskaran, M.M., Wohlford, D., Bastoul, C., Lethin, R.: A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction. In: Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU), pp. 51–61 (2010)
27. Liao, S.W., Du, Z., Wu, G., Lueh, G.Y.: Data and computation transformations for brook streaming applications on multiprocessors. In: Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 196–207 (2006)
28. Liu, W., Vinter, B.: An efficient gpu general sparse matrix-matrix multiplication for irregular data. In: Proceedings of the IEEE 28th International Symposium on Parallel Distributed Processing, IPDPS14 (2014)
29. Murthy, G., Ravishankar, M., Baskaran, M., Sadayappan, P.: Optimal loop unrolling for gpgpu programs. In: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pp. 1–11 (2010). DOI 10.1109/IPDPS.2010.5470423
30. Nugteren, C.: Improving the programmability of gpu architectures. Phd thesis, Department of Electrical Engineering, Eindhoven University of Technology (2014)
31. NVIDIA: Nvidias next generation cuda computer architecture kepler gk110. Whitepaper, NVIDIA Corporation (2013)
32. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara 95050, USA: CUDA C Best Practices Guide, 4.0 edn. (2011)
33. NVIDIA Corporation: NVIDIA CUDA C Programming Guide (2011)
34. OpenMP: The openmp@api specification for parallel programming (2013). URL <http://openmp.org/wp/>
35. Peercy, M., Segal, M., Gerstmann, D.: A performance-oriented data parallel virtual machine for GPUs. In: SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches, p. 184. ACM, New York, NY, USA (2006)
36. PGI: Portland group (2013). URL <http://www.pgroup.com/resources/accel.htm>
37. Ryoo, S., Rodrigues, C.I., Bagsorkhi, S.S., Stone, S.S., Kirk, D.B., mei W. Hwu, W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: PPOPP, pp. 73–82 (2008)
38. of Standards, N.I., Technology: Text file formats. <http://math.nist.gov/MatrixMarket/formats.html>. Accessed: 19 November 2014
39. Tojo, N., Tanabe, K., Matsuzaki, H.: Program conversion apparatus and computer readable medium (2014). US Patent 8,732,684
40. Ueng, S.Z., Lathara, M., Bagsorkhi, S.S., Hwu, W.M.W.: Cuda-lite: Reducing gpu programming complexity. In: Amaral, J.N. (ed.) Languages and Compilers for Parallel Computing, pp. 1–15. Springer-Verlag, Berlin, Heidelberg (2008). DOI 10.1007/978-3-540-89740-8\_1
41. Volkov, V., Demmel, J.: Benchmarking gpus to tune dense linear algebra. In: Proceedings of the ACM/IEEE Conference on High Performance Computing, p. 31 (2008)

42. Wakatani, A.: Effectiveness of a strip-mining approach for vq image coding using gpgpu implementation. In: Image and Vision Computing New Zealand, 2009. IVCNZ '09. 24th International Conference, pp. 35–38 (2009). DOI 10.1109/IVCNZ.2009.5378382
43. Wang, G.: Coordinate strip-mining and kernel fusion to lower power consumption on gpu. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2011, pp. 1–4 (2011). DOI 10.1109/DATE.2011.5763317
44. Wilt, N.: The CUDA Handbook: A Comprehensive Guide to GPU Programming. Addison-Wesley (2013)
45. Xiao, S., chun Feng, W.: Inter-block gpu communication via fast barrier synchronization. In: IPDPS, pp. 1–12 (2010)
46. Xu, Q., Jeon, H., Annavaram, M.: Graph processing on gpus: Where are the bottlenecks? In: Workload Characterization (IISWC), 2014 IEEE International Symposium on, pp. 140–149 (2014). DOI 10.1109/IISWC.2014.6983053
47. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A gpgpu compiler for memory optimization and parallelism management. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 86–97 (2010)
48. Yang, Y., Zhou, H.: The implementation of a high performance gpgpu compiler. International Journal of Parallel Programming **41**(6), 768–781 (2013)
49. Ye, D., Titov, A., Kindratenko, V., Ufimtsev, I., Martinez, T.: Porting optimized gpu kernels to a multi-core cpu: Computational quantum chemistry application example. In: Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on, pp. 72–75 (2011). DOI 10.1109/SAAHPC.2011.8