Computing (2020) 102:977–1003
https://doi.org/10.1007/s00607-019-00744-1

A review of CUDA optimization techniques and tools for structured grid computing

Mayez A. Al-Mouhamed[1] · **Ayaz H. Khan**[2] · **Nazeeruddin Mohammad**[3]

Received: 18 March 2019 / Accepted: 13 July 2019 / Published online: 26 July 2019

Abstract

Recent advances in GPUs opened a new opportunity in harnessing their computing power for general purpose computing. CUDA, an extension to C programming, is developed for programming NVIDIA GPUs. However, efficiently programming GPUs using CUDA is very tedious and error prone even for the expert programmers. Programmer has to optimize the resource occupancy and manage the data transfers between host and GPU, and across the memory system. This paper presents the basic architectural optimizations and explore their implementations in research and industry compilers. The focus of the presented review is on accelerating computational science applications such as the class of structured grid computation (SGC). It also discusses the mismatch between current compiler techniques and the requirements for implementing efficient iterative linear solvers. It explores the approaches used by computational scientists to program SGCs. Finally, a set of tools with the main optimization functionalities for an integrated library are proposed to ease the process of defining complex SGC data structure and optimizing solver code using intelligent high-level interface and domain specific annotations.

Keywords Scientific simulations · Structured grid computing (SGC) · CUDA · Massively parallel programming · Kernel optimizations

✉ Ayaz H. Khan
ayaz.hassan@pafkiet.edu.pk

Mayez A. Al-Mouhamed
mayez@kfupm.edu.sa

Nazeeruddin Mohammad
nmohammad@pmu.edu.sa

[1] Computer Engineering Department, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia

[2] Computer Science Department, Karachi Institute of Economics and Technology, Karachi, Pakistan

[3] Computer Engineering Department, Prince Mohammad Bin Fahd University, AlKhobar, Saudi Arabia

Mathematics Subject Classification 68U20 · 65Y05

1 Introduction

Recent advances in Graphic Processing Units (GPUs) has opened a new challenge in harnessing their computing power as a new general purpose computing paradigm. GPUs have obtained prominence through implementing efficient massive multi-threading as the main strategy for latency hiding. 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. The strategy is to overlap long-latency loads of stalled threads with useful computation in other threads [33]. The Compute Unified Device Architecture (CUDA) is a C-like interface proposed for programming NVIDIA GPUs. However, porting applications to CUDA remains a challenge. CUDA places on the programmer the burden of packaging GPU code in separate functions, explicit management of data transfers between the host and GPU memories, and manual optimization of GPU memory utilization [20]. Hence, the pre-condition to efficiently utilize the GPU resources is a comprehensive understanding of the underlying architecture, and the exertion of complex kernel optimizations. These difficulties have motivated many researchers to develop high-level compilers that restructure the code into optimized CUDA program using a set of loop transformations or optimizations to meet the GPU architectural constraints. Such high level compilers may greatly simplify programming of GPUs, thereby spreading the use of GPUs for supercomputing and scientific computing applications where the tremendous GPU computing power is most needed.

A survey of a wide set of high-level frameworks [45] known as algorithmic skeletons is available online. In some cases, modular programming is used by representing the computation using a graph of modules. The ultimate aim is to develop an explicit programming paradigm with implicit communication among processes. In other cases, a pattern-oriented programming is developed. Other options use parametric skeleton objects for skeleton programming. The skeletal parallel programming framework is useful for data-parallel applications targeting heterogeneous multi-core platforms. Overall, most of presented frameworks, extensions and libraries target only shared (SMP) and/or distributed (DMS) memory parallel computing systems except the FastFlow framework that extends its application to CUDA programming.

1.1 Compilers and code restructuring tools

There is a growing research for reducing the complexity of generating optimized CUDA kernels [11,29,35]. Even though the programming model of CUDA offer a more programmer friendly interface, programming GPUs is still considered error-prone and complex even for expert programmers, in comparison to programming CPUs using parallel programming models, such as OpenMP [13,34]. Most recently, quite a few directive-based GPU programming models have been proposed from both the research community (hiCUDA [20], OpenMPC [27], etc.) and industry (PGI Acceler-

ator [36], HMPP [40], R-Stream [28], OpenACC, OpenMP for Accelerators [8], etc.). In directive-based programming the user provides hints to the compiler to trigger some selective code optimizations with reasonable background on the GPU architecture and the application. On the surface, these models appear to offer different levels of abstraction and reduction in programming effort for code restructuring and optimization.

CUDA-lite [42] proposed a set of annotations to maximize the efficiency of the transformation such as inserting shared memory variables, loop tiling and memory coalesced loads/stores by replacing global memory access with corresponding shared memory access. CUDA-lite allows users to input a naïve CUDA code that treats the memory as a single entity thereby hiding the complexity of handling hierarchical memory. hiCUDA [20] is a directive based language for programming GPUs. The aim is not to automate program optimizations rather it makes the programming in CUDA easier. It still depends on explicit optimizations by the programmer, such as utilizing shared memory or constant memory.

OpenMPC [27] proposed a set of directives to be considered along with OpenMP directives [8]. It optimizes data movement between CPU and GPU by applying the inter-procedural dataflow analysis. 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.

OpenACC is the first standardization effort towards a directive-based, general accelerator programming model portable across device types and compiler vendors. PGI accelerator programming model [36] is a directive based model targeting general hardware accelerators. It currently supports only CUDA GPUs based on OpenACC standards. With light exposure to GPU architecture, the user needs to insert directives into the host program to guide the compiler for particular set of kernel optimizations and code transformations.

HMPP [9] is another directive-based with very high-level abstraction on GPU programming similar to PGI accelerator. HMPP model is based on the concept of codelets that can be remotely executed on hardware accelerators like GPUs. Codelets are offloaded to GPUs based on some manual modification of code structures.

R-Stream [28] is a high-level, architecture-independent programming model that is based on the polyhedral model [10]. It targets various architectures, such as STI Cell, SMP with OpenMP directives, Tilera, and CUDA GPUs. R-Stream performs affine scheduling to extract fine-grained and coarse-grained parallelism. It also performs a set of loop tranformations such as global memory coalescing, loop interchange, strip-mining, loop fusion, shared memory promotion and tiling.

RT-CUDA [25] 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. RT-CUDA converts a C-Like program into an optimized CUDA program by aligning the code to meet the major GPU constraints. A configuration file is used to store hints on selective code transformation. APIs have been added to allow the invocation of external library calls. Finally, a generic program parametrization is used to apply auto-tuning, which will help finding a suitable setting of the resource occupancy.

CUDA-CHiLL [26] is a command based transformation compiler that performs a recipe that contains a set of commands for each required code transformation. Opti-

mization heuristics are applied manually such as the dependences and parallelization, global memory coalescing, shared memory and bank conflicts, and maximize reuse in registers.

The FastFlow framework [2] addresses computations that can be represented by a set of iterative data parallel kernels. Specifically, a loop-of-stencil-reduce is developed to simplify the programming of data parallel programs on heterogeneous multi-core platforms. FastFlow has been used for implementing the Helmholtz PDEs and for streaming Sobel edge detection. For this a 2D stencil is used by combining a 3-by-3 neighbors in updating the solution matrix. The work extends the earlier work on the SKePU programming framework [15]. FastFlow presents an important parallel program optimization for implementing the reduce operations (map, reduce, map-reduce and stencil reduce) in a heterogeneous GPU systems.

Section 2 presents a summary of current compilers and restructuring tools applying various GPU optimizations. It also discusses the GPU optimizations that are required to enhance resource utilization, a pre-requisite for GPU performance.

Most of the proposed computational paradigms aim at determining the involved stencils in the solver equations based on the knowledge of the PDEs involved in the physical model. In this view our proposed computational paradigms and that of FastFlow positively help developing fast prototyping altogether with optimizing the implementation of the most time consuming part that is the stencil involved in the solver system.

1.2 GPUs and structured grid computing (SGC)

The GPU tremendous computing power is extremely useful for accelerating many science applications that are based on discrete numerical simulation. Extracting parallelism depends on the application nature. For example, in signal processing the multi-dimensional digital filter matrices are determined based on dependence graph analysis (DAGs), divide-and-conquer strategies and pipelining [18].

Structured grid computation (SGC) is one important class of science applications [19,22,30,32,41,43]. Hence, there is a great interest to develop systematic optimization approaches for accelerating SGCs using GPUs. Generally, SGCs consist of repeatedly solving a sparse system of linear equations using an iterative linear algebra solver (ILAS) algorithm that represent the solver. Most of the simulation time is spend within the solver, which justify the need for an efficient solver implementation. The most time consuming part of a solver is the sparse matrix–vector multiplication (SpMV) operations [22,30,44,48]. Sparse matrix and vector calculus attracted great attention. For example, optimizing the storage of sparce matrices and the implied matrix–vector calculus is one critical issue in the efficient implementation of SGCs on GPUs. In most cases, users have standard storage schemes that are adequate for general sparse matrices with quasi-random rdata layout. In many cases, the above issues are manually addressed due to lack of effective tools especially when there is need to efficiently exploit the regularity and the data pattern in the distribution on non-zero elements to increase problem size and simulation accuracy.

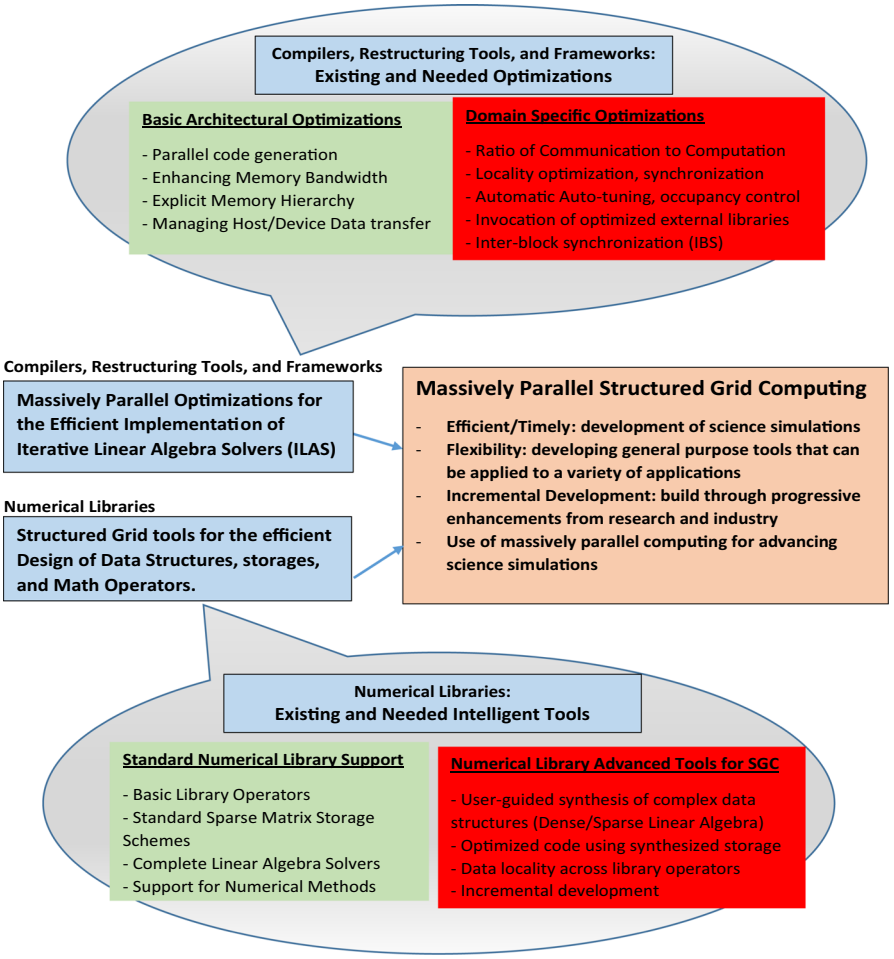

Fig. 1 An overview of required optimizations to support SGC. The optimizations that are briefly addressed by existing tools are marked in green; the optimizations that are mostly not addressed by the existing tools are marked in red (color figure online)

1.3 Paper contributions and organization

Many computational science applications can be discretized and simulated using SGC approaches. The objective of this paper is to explore the techniques that accelerate the implementation of efficient science simulations for the class of SGCs applications. It identifies SGC simulations that can be accelerated by "basic machine dependent optimizations" and "domain specific optimizations & tools". Figure 1 summarizes the basic idea of this paper.

An efficient implementation of SGC on GPUs must account for two fundamental class of optimizations, which are (1) GPU machine dependent architectural optimizations and (2) SGC domain specific optimizations, see Sects. 2.1 and 2.2 for details.

Due to their complexity and multi-objective nature, the former optimizations have been only partially addressed in proposed compilers and restructuring tools. The later optimizations represent the functionalities needed for implementing iterative linear algebra solvers using intelligent tools for developing optimized storages, sparse matrix data structures, and implied matrix and vector operators. Section 2 presents more details about the compilers and restructuring tools which will enlighten the reader about how the above optimizations are implemented.

NLs helped the scientific community by providing optimized parallel/GPU code for: (1) a rich variety of operator based vector/matrix functionalities; (2) few standard sparse matrix storage schemes (COO, CSR, HYB, DIA, etc.); (3) few linear algebra solvers (Ax=b) with some preconditioning such as Jacobi, SOR, BiCG, and GMRES; (4) few numerical methods such as matrix Factorization, Gaussian elimination with pivoting, LU, QR, and LSE. However, the following much needed enhancements for SGC are missing: (1) User-guided synthesis of complex data structures; (2) Optimized code using synthesized storage; (3) Data locality across library operators; (4) Automatic auto-tuning.

Section 4 details a set of required enhancements and intelligent tools to help scientists describe their complex data structures and automatically generate optimized code for quick prototyping of SGCs.

For relevant background and basic terminology related to the GPU architecture, heterogeneous programming details and the working of linear algebra solvers, the readers can refer to references [1,3,24,25].

The rest of the paper is organized as follows. Section 2 explores the basic GPU optimizations, how these optimizations were addressed in research and industry compilers, and current practices for programming linear algebra solvers. Section 3 introduces SGC with an example of oil reservoir simulation. It discusses SGC implementation strategy, and review the libraries, tools and optimizations used in these implementations. Section 4 explores how computational scientists define SGC data structures and optimize the solver algorithm, which will help us identify the key optimization functionalities for an integrated SGC library that will ease the process of designing complex SGC simulations. Section 5 presents our incremental contributions towards the integrated SGC library.

2 GPU optimizations for linear algebra solvers

This section presents the basic GPU architectural optimizations (BA), the domain specific compiler optimizations (DS), and how research and industry GPU compilers have implemented transformations to enhance code efficiency. It also identifies the optimizations needed for the class of Iterative Linear Algebra Solvers (ILAS) and explore how scientists program these solvers.

2.1 Basic GPU architectural optimizations

The GPU architecture and its execution model provide detailed information on how the GPU optimizations must be utilized to achieve the best possible application performance. Following is a list of *Basic Architectural Optimizations* (BAs) and their functional specifications that must be applied by the software tool or the compiler to generate efficient CUDA programs:

1. The Parallel Memory Bandwidth (PMB): This aims at mapping threads within a warp (group of threads run in lock-step) to access data from distinct storages in the device memory. The compilers/tools must explore different correct mappings for the addresses generated by neighboring threads and select a mapping that guarantees coalesced access to global memory. For shared memory accesses, threads within a warp must map their accesses into distinct memory banks to avoid serialization. Data access requests to global memory can be reordered in parallel by multiple channels and banks. However, the memory bandwidth is efficiently utilized when the accesses to the memory channels are balanced, without congested channels.
2. The locality optimization (LO): A GPU has several streaming multiprocessors (SM), each has a small fast shared-memory (ShM). Sharing among SMs is done using a large slow global memory (GM). The locality optimization aims at enhancing the use of the deep explicit GPU memory hierarchy by using four main actions. The first step consists of copying data once into ShM to maximize data reuse while maintaining a data footprint that meets memory constraints. The second step converts the original loop using the technique of blocking or tiling with a fixed maximum size to fit in the ShM capacity. The third step consists of making an efficient use of the available large register file for temporary data. Finally, use read-only special portions of GM that are the constant and texture by preloading the data in them before entering kernels.
3. The Input/Output GPU Memory Allocation (I/O): The use of inter-procedural dataflow analysis to optimize data movement between host CPU and device 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.
4. Computation Partitioning and Decomposition (CP): It consists of three fundamental actions which are (1) manage block-level and thread-level parallelism, (2) map block/kernel organization and dimension to the data structure of the computation, (3) use of address transformations to map threads to the results and adjust thread granularity to amortize transfer/processing ratio.

2.2 Domain specific compiler optimizations

Although the BA optimizations are essential, they are far from being sufficient to optimize simple domain specific applications. An important target application for restructuring tools is the area of *Iterative Linear Algebra Solvers* (ILAS).

ILAS can benefit from the existence of highly optimized math libraries for basic dense and sparse linear algebra operations. These libraries are developed by the academia and industry communities to help providing code for multi-core and many-core computers for a variety of applications including ILAS. Libraries may have optimized code for many algebra operators that can be invoked from many high-level languages. Library operator calls offer many substantial performance advantages such as sparing the user from direct exposure to the GPU details, in addition to rapid prototyping and code portability. Further, libraries are constantly enhanced and new features are added.

To efficiently implement ILAS algorithms, a restructuring tool must embody the BA optimizations in addition to the ability to efficiently implement some domain specific (DS) optimizations. For this the following additional DS features should be integrated in addition to the aforementioned BA optimizations:

1. Inter-Block Synchronization (SYN) is needed because of the iterative nature of ILAS algorithms. Here, threads cannot start the next iteration before making sure all threads have completed the current iteration. Since GPUs offer no global synchronization, there is need for a customized inter-block synchronization mechanism, when exact algorithm behavior is needed to ensure correctness. Some of the proposed synchronization techniques are: (1) kernel entry/exit, (2) lock-based, (3) lock free, (4) relaxed synchronization, or (5) adapt synchronization to algorithm depending on expected degree of thread load unbalancing.
2. Invocation of Optimized External Libraries (IEB): Some external libraries have been optimized at lower level programming and may deliver substantial performance advantages over manually optimized regular code. Efficient invocation of external libraries require full understanding of its parameters and related implementation logic to select proper parameter values.
3. Optimization of Architectural Parameters (AP): Due to many GPU occupancy constraints, there is a need to carry out some resource management analysis and find out the most suitable machine occupancy parameters. Empirically searching in a space of possible configurations using code parametrization and auto-tuning techniques allows finding the optimal values of kernel parameters for best performance.

Following subsections explore the research issues for each DS optimizations and discuss some potential solutions.

2.2.1 Global synchronization

The Lock-based synchronization uses atomic operations on global variables defined in GM. When all threads of a block finish their work, the first thread of each block atomically decrements a global variable (mutex) and continues checking as long as it is more than zero. The drawback is the hot-spot in polling of GM by the terminating block. In Lock-free [37,47], each terminating block b sets its entry in a global input array $Ain(b)$ to post its termination. Next, the thread checks the completion of other blocks using the other block locations of Ain. Note that access to $Ain(b)$ need not be

atomic because $Ain(b)$ can be set only by block b. The barrier is passed when a block finds that all entry of Ain are set.

Relaxed synchronization (RS) [25] allows two iterations to overlap in time. A completing thread-block stores its range of results in GM and starts the next iteration by using the partial results from the other threads. A global array is used to collect the results from completing thread-blocks. The thread-block terminates the current iteration when it has processed all partial results in the current operation. This approach is profitable when there is enough load unbalancing among the threads to offset the overhead of processing partial results.

Another interesting scheme is the Re-Ordered Synchronization (ROS), which was proposed to hide the global synchronization overhead by allowing a completing thread to execute some independent work that must be done anyway. ROS consists of re-ordering the operations so that a completing thread block stores its partial results in GM and starts an independent operation to avoid polling GM for the completion of other blocks. Two global arrays *Ain* and *Aout* are used to post block completion and to check completion of a global reduction respectively. The thread block that was last in posting its completion carries out the reduction of partial results in a global variable and sets all entries of *Aout*. ROS advantages are: (1) the synchronization time due to load unbalancing is hidden by some computations, and (2) eliminate the need for atomic access to global flags as well as polling for the other thread completions.

2.2.2 Optimization of architectural parameters

In GPU, all data movements among the cache memory hierarchy are highly dependent on the CUDA kernel structure as there is no cache coherency implemented within the GPU architecture. In addition, it is difficult to determine the optimal parameters that define the GPU machine occupancy. These parameters are the grid block size, the thread block size and tile granularity. Usually these parameters are found using manual empirical testing or using a tool like OpenTuner [5], which is a very time-consuming process due to many possible combinations. Therefore, the need for a compiler auto-tuning approach to evaluate the performance of a newly generated parametric kernel with various possible combinations of the above occupancy parameters. The pruning of the list of possible parameters is used at three levels to reduce the repeated compilation and execution of the kernel [24]. The three levels of pruning consists of skipping those tile sizes which do not equally distribute (1) the number of resultant elements among all threads (array block), (2) among all kernel blocks (Kernel block level), or (3) parameters which require more than the available registers (active block level). For each combination, the number of registers/thread and shared-memory (ShM) per block are determined. Next the number of Active Blocks by Warp (ABW), Active Blocks by Shared Memory (ABShM), and Active Blocks by Registers (ABR) are calculated. Parameter pruning is carried out at the Active Block Level and generates a list of possible optimal parameters. Finally, the kernel is run for each combination of parameters in candidate parameter list and the optimal combination of parameters that give the minimum execution time is retained.

2.2.3 Challenges for numerical libraries

Numerical libraries have made significant progress with respect to code versatility, operator diversity, and ready-made solvers in many cases. However, just implementing an ILAS using numerical library calls from a high-level language may not work because of the following two reasons. (i) The provided sparse matrix storage schemes are appropriate to handle a class of sparse matrices without much regularity. There is a further need to adapt the storage scheme to take advantage of the non-zero pattern regularity, block structure and other specific features. (ii) The optimized operator library calls assume their operands in global memory (GM) to gain generality. This offsets the benefit of data locality when chaining the solver operators. Thus, the operator data locality is lost unless a more general operator semantic is developed to ensure all possible operator chaining be done at the level of the shared memory.

The application of stencil based relationships on structured grids results in some problem-specific features such as the solver matrix layout, the sparsity pattern, and the number of state variables to be updated at each grid point. Numerical libraries have a variety of optimized sparse matrix storage schemes, which aim at minimizing sparse matrix storage and enhancing the computation of basic algebra operators. The use of NL for SGC implementation is useful for generic sparse data structure and the matrix–vector math operators. However, NL do not have tools that takes as input the stencil-based relationships in a given SGC and determine the sparse matrix data layout and its optimized storage scheme. Hence, the scientists are responsible for the tedious manual work or use standard storages that cause significant drop in performance. Another important issue is that sometimes NL does not produce optimized code because of the lack of tools that exploit the data locality across a chain of operator invocation, i.e. locality is lost from one operator invocation to the next. Hence, the performance of library functions is generally far from that of a manually optimized code.

Hence, there is mismatch between the computational power of GPUs and the degree of SGC optimizations when solely using library implementations. To compile efficient library code, there is a need for an analysis of the strategies used in current compilers. Proper analysis allows library users to identify the missing constructs for efficiently implementing iterative solvers, and achieve high-degree of kernel optimization. Another problem is how to let the user describe the features of the domain specific sparse matrix to enable the efficient implementation of the SpMV, sparse matrix storage and the solver [30]. Finally, the evolving GPU architecture requires some reflections on a integrated library framework to provide portable, flexible and viable solutions.

2.2.4 Summary about optimizations in research and industry compilers

The discussion from the previous sections shows how research compilers have addressed the optimizations needed to take advantage of the massive parallelism in GPUs. Table 1 shows how all the aforementioned BA and DS optimizations (see Sects. 2.1 and 2.2) have been addressed in the available software frameworks and compilers. This table is built based on the understanding of the published description of optimizations used in these compilers and frameworks. The main feature of these compilers is that they present a simpler GPU programming model. However, the opti-

Table 1 Comparison of software compilers and frameworks in terms of optimization specifications

Optimization specifications	CUDA-lite	hiCUDA	OpenMPC	PGI	OpenACC	HMPP	R-Stream	CUDA-CHiLL
Input/Output GPU Memory Allocation	None	Medium	Medium	Medium	Medium	Medium	High	Low
Computation Partitioning and Decomposition	None	Medium	Medium	Medium	Medium	Medium	High	Low
Locality optimizations and Datacopy Transformations	High	Medium	Medium	Medium	Low	Medium	High	Low
Parallel Memory Bandwidth	High	High	High	High	High	High	High	High
Optimization of Architectural Parameters	None	None	None	Low	None	Low	Medium	Medium
Use of automatic compiler optimization and/or programmer-guided optimization	Medium	High	Medium	Medium	High	Medium	High	High
Synchronization across SMs	None	None	None	None	None	None	None	None
Invocation of Optimized external Libraries	None	None	None	None	None	None	None	None

mizations needed to generate tailored kernels for scientific simulations are missing. The complexity of finding systematic and automatic GPU optimizations makes these compilers less efficient than manually optimized programs for general purpose computing. The following are the major limitations for the efficient implementation of ILAS algorithms using parallel compilers and libraries:

1. There is a semantic gap between current GPU compilers and the optimizations needed for ILAS, which are far from meeting the expectation of raw numerical algorithms. For example, lack of global synchronization and absence of automatic auto-tuning tools degrade resource utilization.
2. The code is not optimized to take advantage of the operator data locality in a sequence of library calls as it always refer to the data stored in the lower levels of the memory hierarchy.
3. Numerical libraries have standard sparse matrix storage schemes. Most of the sparse matrices found in science simulation have domain specific data structures. Libraries and programming tools fail to capture the regularity in sparse data structures and synthesize customized storages.
4. ILAS computing suffered in the past from the lack of powerful tools to enable scalable implementation on cluster computers. This gap is becoming wider with many-core technology due to complexity of adapting ILAS to massive arithmetic parallelism, the explicit memory system, and the multi-threading strategy.

As a result it is rare to find an application in the area of linear algebra solvers that has been significantly accelerated using the above compilers. The next section explores how SGC researchers have been using parallel computers, software tools, numerical libraries, to develop structured grid simulation. It focuses on the difficulties in the above process and identify some of the needed tools to speedup the development of SG applications on GPUs.

3 From science simulation (SS) to structured grid computing (SGC)

Science simulation (SS) is specially useful to help emulating the design process, which presents a significant cost saving in a variety of research and engineering areas. A physical process is modeled using a set of partial differential equations (PDEs). Discretization of the PDEs lead to representing the problem using a large 3D grid of cells with each grid cell having a set of states. Most of the models are inherently non-linear. In any given simulation, each grid cell has a number of independent state variables. A grid cell interacts with its neighbors through physical exchanges of the state variables. The stencil defines how the state of neighboring cells interact with each other according to the PDE laws. Generally the physical process is non-linear and linearization is done by using the Newton-Raphson method.

To represent the interaction among all the grid cells, a 3D grid having N cells, each cell has k state variables, is unfolded into 1D representation having kN variables by using a simple address mapping function. This is useful to build the solver matrix. Each cell is represented by its k state variables in the 1D representation. A 2D Jacobian matrix (solver) is built by mapping the 1D unfolded grid onto the rows and the columns,

e.g. element (i,j) represents the interaction coefficient between grid cells i and j. Hence, the interaction among the grid cells is represented by a Jacobian matrix A of size (Nk)x(Nk). Generally, matrix A is sparse because a cell interact only with its immediate neighbors. This consists of repeatedly solving a system of linear equations of the form Ax=b to approximate a non-linear solution, where A is a square sparse matrix, b is known vector, and x is the unknown vector. Solving this system means finding solution x that satisfies Ax=b. Solution x is used to update the model which in turn updates the value of matrix A and vector b. The process of solving for x and updating the model continues until reaching some converging condition.

There are many linear algebra solvers which solve Ax=b using direct and iterative approaches. The LU factorization is one example of a direct method, and the Jacobi, Conjugate Gradient (CG) and GMRES are examples of iterative approaches. A science simulation spends most of its running time in solving the above system of linear equations. Thus, most of the optimizations focus on the solver which includes the main algebra operators like the Sparse Matrix–Vector Multiply (SpMV), Matrix-Matrix multiply (MM), inner product of two vectors, addition/subtraction of matrices and vectors, etc. Efficient synchronization among all the working threads is another important optimization to produce correct solutions.

The research and industry communities have developed compilers and numerical libraries to help alleviate the complexity of programming directly in CUDA. However, due to the complexity of finding systematic and automatic GPU optimizations and efficient implementation of linear algebra operators make these approaches much less efficiently implemented than manual optimization. This is widening the gap between the performance of science simulation and the large computational capabilities of GPUs. Table 2 shows the summary of SGC research based on code optimization (CO), numerical libraries (NL), and/or data structures (DS) for sparse matrices.

4 Integrated SGC library (ISL)

GPUs have a great potential for scientific simulations. Utilization of a significant fraction of GPU peak performance is needed to enable enhancing the accuracy in SGC simulations. This task is quite challenging because of the GPU architectural complexity and the lack of efficient tools to customize the SGC data structures and to adapt the code to the algorithm properties. Hence, there have been various research efforts to help users in GPU programming, but a comprehensive solution that efficiently addresses the SGC data structures and algorithms is still in a research phase. So, there is a need of an Integrated SGC Library (ISL) for scientific simulations that will efficiently support scientists in GPU programming and relieve them from the tedious task of programming their solver matrices, customizing optimized storages, redesigning operators to preserve data locality, and efficiently embedding the code using architecture-specific GPU optimization details. ISL can be seen as an intelligent interface for describing PDEs and structured grids at a very high-level using a notation that resembles mathematical formulas. An attempt in this direction is found in the UFL (Unified Form Language) [4] which allows users to describe finite element equations that are translated into kernels by FEniCS Form Compiler (FFC). ISL needs

Table 2 Summary of SGC Research work that is based on code optimization (CO), use of optimized operators and tools in Numerical Libraries (NL), or use/develop storage schemes and data structures for sparse matrices (DS)

Paper title	Employed optimizations	Brief details of manual optimizations used for performance enhancement
Stencil-aware GPU optimization of iterative solvers [30]	DS, CO, NL	Synthesized an optimized storage scheme to store non-zero data and the used object-oriented data structures for SpMV. The values of GPU structural parameters for the synthesized storage were optimized using R-CUDA tuning tool
A generalized framework for auto-tuning stencil computations [22]	CO, NL	Packaging generalized stencil kernels as libraries and finding combination of tunable parameters that maximizes computational efficiency for a given algorithmic kernel
Towards dense linear algebra for hybrid GPU accelerated many-core systems [41]	CO, NL	Splitting a computation to fully exploit the power of the hybrid many-core components and application of Dense Linear Algebra (DLA) such as LU-factorization algorithm
Optimizing stencil computations for NVIDIA Kepler GPUs [32]	CO	Use of techniques for stencil computations for regular grids to enhance data locality with shared-memory combined with warp specialization for higher instruction throughput
High-performance sparse matrix–vector multiplication on GPUs for structured grid computations [19]	DS, CO	A sparse matrix storage scheme that takes advantage of the diagonal structure without explicitly representing many zero elements in the sparse matrix (compared to DIA) and build corresponding SpMV.
Benchmarking GPUs to tune dense linear algebra [43]	CO	Exploit blocking in hybrid CPU-GPU, increasing parallelism and regularity in the problem that provide slightly higher performance with enhancement of MM (GEMM), LU, QR and Cholesky factorizations
Optimizing sparse matrix–vector multiplication on CUDA [44]	DS, CO	SpMV optimized CSR storage format, optimized threads mapping, and avoiding divergence judgment
An Improved Sparse Matrix–Vector Multiplication Kernel for Solving Modified Equation in Large Scale Power Flow Calculation on CUDA [48]	DS	Enhancing SpMV using a new (ICSR) storage format to solve the problem of global memory alignment
An auto-tuning framework for parallel multicore stencil computations [23]	CO	A stencil auto-tuning framework to avoid limitation to single kernel instantiations and the difficulty of assembling different kernels into a library

Table 2 continued

Paper title	Employed optimizations	Brief details of manual optimizations used for performance enhancement
Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures [14]	CO	Built an auto-tuning environment that searches optimizing parameters to minimize runtime, while maximizing performance portability over a variety of platforms for stencil (nearest-neighbor) computations.
Optimizing the matrix multiplication using strassen and winograd algorithms with limited recursions on many-core [24]	CO	Implement a depth-first approach for Strassen/Winograd algorithm for dense matrix multiply and shows that a few recursion levels are sufficient to outperform best known times but with some loss of accuracy due to trading matrix addition instead of multiplications
Efficient CSR-Based Sparse Matrix–Vector Multiplication on GPU [17]	DS	Alleviate the limited memory bandwidth and data locality of SpMV using an adaptive multi-level blocking, compression of column indices and the reuse of input vector elements. Auto-tuning is used to find the best set of parameters by estimating the memory traffic and predicting the performance
Automatic Selection of Sparse Matrix Representation on GPUs [39]	DS	Analysis of the inter-relation between GPU architecture, sparse matrix representation and the sparse dataset using a set of 700 matrices with different sparsity features. A machine learning decision model automatically selects the best representation on a given target platform, based on the sparse matrix features
Characterizing Dataset Dependence for Sparse Matrix–Vector Multiplication on GPUs [38]	DS	Study of statistical features of CuSPARSE storages like CSR, ELL, COO and ELL-COO scheme for 27 matrices and attempt to correlate performance with each representation with simple aggregate metrics
Efficient CSR-Based Sparse Matrix–Vector Multiplication on GPU [17]	DS	Enhancing the CSR sparse matrix storage scheme for GPU using dynamic assignment of different numbers of rows to each thread block on the basis of the number of rows involved for each block, which allows coalesced access to the global memory. Evaluation shows favorable results compared to CSR and adaptive-CSR on the C2050 and K20c GPUs

Table 2 continued

Paper title	Employed optimizations	Brief details of manual optimizations used for performance enhancement
Optimization of sparse matrix–vector multiplication on emerging multicore platforms [46]	CO	Analysis of SpMV optimization strategies across a broad spectrum of multicore environment and provide key insights into the architectural tradeoffs of leading design strategies, in the context of demanding memory-bound numerical algorithms
Inter-block GPU communication via fast barrier synchronization [47]	CO	To reduce overhead in CPU-based synchronization (kernel exit), a GPU inter-block synchronization is proposed using lock and lock-free approaches. Micro-benchmarking of the FTT, dynamic programming, and bitonic sort shows that lock-free schemes have the least overhead
Implementing sparse matrix–vector multiplication on throughput-oriented processors [7]	DS, CO, NL	To alleviate the irregular memory to broad spectrum of sparse matrices it is proposed to choose the storage format such as DIA, ELL, CSR, and COO, use of fine-grained parallelism and impose sufficient regularity on execution paths and memory access patterns, design kernels with minimize divergence
A new method of sparse matrix–vector multiplication on GPU [21]	DS, CO, NL	A non-parametric, self-tunable, approach to data representation of SpMV for power-law graphs. Using real web graph data coupled with a tiling algorithm, can yield significant benefits over the various GPU implementations on a number of core data mining algorithms such as PageRank, HITS and Random Walk with Restart
High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning [12]	CO, NL	Study of Conjugate Gradient (CG) solver for sparse matrices on a GPU-cluster where faster communication is needed to achieve scalability. A hierarchical hypergraph-partitioning is used for communication reduction and load balancing over a heterogeneous system
Accelerating the solution of families of shifted linear systems with cuda [16]	CO, NL	Evaluate the open source GPU coding to solve shifted families of sparse linear systems for the multi-mass conjugate gradient (CG-M) and multi-mass bi-conjugate gradient stabilized (BiCGStab-M) methods, which are used in lattice gauge theory for simulating dynamical fermions. GPU coding favorably compare to CPU-based coding.

Table 2 continued

Paper title	Employed optimizations	Brief details of manual optimizations used for performance enhancement
Acceleration of GPU-based krylov solvers via data transfer Reduction [6]	CO, NL	Proposed to re-design the SpMV using CuBLAS library to alleviate the communication overhead when implementing Krylov iterative methods (KIM) using traditional numerical libraries; concluded that similar algorithm optimizations are profitable for other KIM solvers
Unified form language: A domain-specific language for weak formulations of partial differential equations [4]	NL	Unified Form Language (UFL) is proposed for representing weak formulations of PDEs. UFL supports variational and functional forms, automatic differentiation, arbitrary function space hierarchies and flexible tensor algebra. UFL expresses finite element methods in near-mathematical notation, resulting in compact, intuitive and readable programs
Auto-tuning stencil-based computations on GPUs in Cluster Computing [31]	CO, DS, NL	The diagonal storage format of Newton-Krylov iterative methods is extended to an efficient blocked data structure using the PETSc parallel numerical toolkit. Proposed a tunable parametric implementation to enable automatic search for the best parameters Using Orio framework

to be implemented as a viable software system focusing on ease of extendability and maintainability, as the working data structures and algorithms will evolve and new annotations and optimizations will be required to adapt to changes in the simulated problems and the GPUs. The following are the detailed features of the desired ISL:

1. **Intelligent Data Structure Interface (IDSI)**: There is need to automate the process of building, refining, and generating the sparse matrix data structures, which is the pre-condition to generate optimized programs for scalable solution of PDEs. The process should be based on a mechanism to capture the grid features such as the regularity in the sparse matrix and its pattern. This can be implemented using a set of high-level annotations to help users synthesize the main solver matrix. Users provide guidance to the principal data structures by taking advantage of the problem structure in implementing customized solutions. Given an SGC and a stencil relationship, a set of linear equations (SLEs) corresponding to the application of the stencil to all the grid points can be easily derived. The obtained SLEs are represented by a sparse matrix, which is the solver matrix. Therefore, it is useful to develop an intelligent tool that allows the user to describe the SG and the relevant simulation constraints such as the stencil operator, boundary conditions, initial conditions and cell components.

IDSI tool may use graph properties to build the basic grid cell structure with its computing links to other cells and determine the rules at the grid boundary and their trigger conditions. Using inference rules on the synthesized data structure will enable the automatic generation of the solver matrix for arbitrary grid size. Generated solver matrix with the incorporation of all the constraints identify the matrix non-zero blocks, which might be emerging from the stencil definition, initial conditions, or boundary conditions. In addition, the tool will prepare the link between the matrix structure and the physical model parameters to update the solver data on every iteration of the simulation, a task that is essential when implementing the solver algorithm. This flexible approach will prove its usefulness for assessing the simulation scalability in the process of enhancing the simulation accuracy. Thus, the main benefit of IDSI is that it minimizes the user effort from the task of defining the grid and its constraints, and let the system find generic rules that scale the problem to arbitrary size and handle the complexity of the structure and assessment.

2. **Automatic assessment and selection of a library storage scheme**: There is a need to assess the performance of standard library's storage schemes given the structure of the synthesized solver matrix, its structured regularity and distribution of non-zero element (nze) blocks. A tool is needed for the automatic assessment of the efficiency of library sparse matrix storage schemes. One approach is to select a subset of available storage schemes by comparing the recognized storage profitability features with those found in the current matrix structure. The automatic assessment must account for the total storage required, number of operations needed to compute the index of non-zero elements, ratio of communication-to-computation involving transfer from GPU global memory accesses to shared-memory, bank conflicts at shared-memory, and available bandwidth. A storage scheme can be retained when its implied performance meets the user specified optimization level.
3. **Synthesizing custom storage scheme**: The user needs to guide IDSI for building a custom storage scheme if no library storage scheme may provide an acceptable performance level. An interactive process in which the user is provided with high-level annotations to guide the compression of the non-zero elements in the solver matrix. The objective is to build progressively a storage scheme that balances the matrix storage requirement with the overhead of address calculation and memory access in the basic SpMV operation. The experience with manual storage optimization indicates that many possible storage schemes can be interactively designed such as converting diagonal blocked data structure into columns or rows, collapsing rows or columns of non-overlapping data, caching block address keys to avoid multiple conditional statements in code, clustering irregular blocked structure and use a two-level hybrid compression coding. These techniques have been experienced in different fields and may be used as key user annotations in a bottom-up approach to construct the storage scheme from the smallest near-compact pattern (NCP) to whole matrix by taking advantage of the regularity, graph properties, and repeatability found in the solver matrix among the non-zero blocks. The tool may display the snapshot of the NCPs to help the user visualize the pattern regularity at different grid sizes. A grid plan is bounded by many blocks of zeros, which are due to shrinking the stencil at the plan boundary. The result is a regular distribution

of NCPs, where the NCP size depends on the plane size. Although the NCP may change its pattern depending on grid dimensions, the block connectivity within the NCP is largely preserved. The tool needs to validate the synthesized storage and assess its expected performance by using an evaluation technique such as the SpMV computation.

4. **Optimizing the solver algorithm**: The optimized math operators available in the numerical libraries need to be redesigned to preserve inter-operator data locality in shared-memory and register files, which are currently available through generic library calls. Currently library math operators pick up their operands from the global memory. Different argument scenarios must be available for each operator to ensure the use of the data operand wherever created by its predecessor. Similarly, produced data should be cached wherever appropriate to minimize data motion across the data dependent operators. Currently, most of these data motion optimizations are manually performed. A chained list of operators should be automatically translated, following a global data dependence analysis, into a chain of selected operator scenarios that minimizes data motion. Optimization techniques to account for the GPU architectural features such as the small shared-memory size, coalesced global memory, and conflict avoiding in shared memory should be systematically used in the algorithm implementation. This ensures some acceptable level of communication-to-computation ratio for a given GPU. The code needs to be easily modified for solving (i) larger problems, (ii) fixed size problems but with faster execution, or (iii) fixed size problems but with increased accuracy. The compiler should have sophisticated optimizations to trade memory bound or time bound implementations.
5. **Integrated auto-tuning (IAT)**: Currently SGC kernel auto-tuning is manually done or the application migrated to another environment for a complex user supervised auto-tuning. There is a need for an integrating auto-tuning (IAT) as part of the library development process to hide the complexity of fine tuning code and to avoid exposing the user to GPU intricacies. Auto-tuning need to be redesigned to spare the user from being exposed to process of searching the most optimized combination of architectural parameters, which are complex and highly machine-dependent. IAT should handle many complex and interrelated architectural features such as the automatic generation of parametric solver kernel (PSK) to enable the use of sophisticated auto-tuning techniques. PSK capture the salient GPU occupancy parameters like the kernel structure, thread block size, thread granule size, compilation flags, etc. The parametric code can also be a user guided process, which enables selecting the GPU salient parameters and leave it to the tool to prune unlikely parameter combinations and focus on a small set of values that are assessed using empirical evaluation. Auto-tuning results may change depending on the grid problem size, which favors an integrated approach that benefits from the cumulative knowledge and overall attempts in scaling up the code from one level to the next. IAT increases the portability of the solution as auto-tuning can be done for different architectures without the need for code recompilation.
6. **Dynamic Load balancing (DLB)**: ISL should support DLB to tolerate load unbalancing in iterative algorithms. DLB feature should be scalable and hence should avoid the use of explicit lock-based synchronization. The experience shows that

hiding synchronization overheads by running some computation that must be done any way proved to be a profitable alternative in designing linear algebra solvers. DLB should offer different user selected options for enhanced load balancing such as allowing the overlap of different iterations with proper management, work-queues and work-stealing techniques.

7. **Backward compatibility and extendibility**: ISL should culminate in a viable software system focusing on backward compatibility and ease of maintenance. Specifically, it should be extendable to allow the addition of new data structures, optimizations and solvers that proved to be efficient in optimizing SGCs. For code simplicity, generality and reusability, ISL may be based on object-oriented (OO) design.
8. **Profiling and debugging**: ISL should provide software tools to let users visualize the key bottlenecks in the code. It should facilitate the debugging when code crashes. For this purpose, it should store the core files needed including all stacks at the time of the crash. Debugging can be simulated on a small number of cores interactively.

5 Proposed tools for integrated SGC library (ISL)

ISL is a complex library and cannot be designed at once, but the individual components are designed gradually. This section presents our incremental contributions towards the development of ISL.

5.1 SpMV and BiCG-stab optimization for a class of Hepta-diagonal sparse matrices on GPU

A Structured Grid Development Tool (SGDT) [1] is proposed to customize the design of the solver algorithm for reservoir simulation (FRS) for arbitrary grid size, stencil relationship, number of components, and boundary conditions. SGDT can be summarized as follows:

1. **Generalized Sparse Solver Matrix:** Deriving the generalized hepta matrix GH based on the knowledge of the structured grid (J,H,I) all together with a set of cell components, stencil relationships, initial and boundary conditions, and FRS model for updating the data blocks following each solver iteration. To build the solver matrix, the SG is unfolded into 1D form taking into account the grid dimensions and number of cell components. The offsets from a grid cell to its stencil cells are computed and stored for use in SpMV when indexing the dot product. Taking advantage of SG regularity, the solver matrix compactly stores the NZs as well as a common offset vector for all the rows. This enables the automatic generation of a family of sparse solver matrices as a function of grid dimensions, a number of cell components, stencil and initial and boundary conditions.
2. **Optimizing Sparse Matrix–Vector Multiply:** Each SpMV result is assigned to separate thread. Hence neighboring block of results are implicitly assigned to threads within each warp. SpMV optimization is based on optimizing the sparse

matrix storage and minimizing address calculation by shared an explicit offset vector coalescing memory access and indexing operations, which are the prerequisite for the design of an optimized SpMV CUDA KernelAddress computation is optimized by sharing an explicit offset vector among all threads that are mapped to identical number of SpMV results. Threads use a coalesced access due to row access of matrix NZs and retrieve the multiplicands using the shared offset. SpMV is coded as a parametric module, which enables the use of code auto-tuning. Auto-tuning searches for a combination of GPU architectural parameters (grid, thread blocks, thread granule size, and other compiler flags) for final optimization of the SpMV CUDA code.

3. **Optimizing the Solver:** Most iterative solvers can be expressed using vector, vector-matrix, and vector scaling operations. The solver dependence graph helps in orchestrating the following parallelization steps: (1) identify and group global synchronization points, (2) implement optimized operators like SpMV over the synthesized sparse matrix, inner product, vector addition and scaling, and convergence condition, and (3) enhance vector data locality. Similarly, auto-tuning is carried out over the parametric solver code as a final optimization step.

For the forward petroleum oil and gas reservoir simulation, the application of a stencil relationship to structured grid leads to a family of generalized Hepta-diagonal solver matrices with some regularity and structural uniqueness [1]. A customized storage scheme that takes advantage of generalized Hepta-diagonal (GHD) sparse pattern and stencil regularity is proposed. The invocation of numerical libraries operators is made using multiple-kernels invocation, which causes loss of data locality due to kernel exit and re-entry. An in-kernel execution model (IKEM) is proposed based on a lock-free inter-block synchronization. Thread blocks are assigned some independent computations to avoid repeatedly polling the global memory. Other optimizations enable combining reductions and collective write operations to the memory. IKEM allow preserving vector data locality and avoiding saving vector data back to memory and re-loading on each kernel exit and re-entry. IKEM is suitable for many iterative solvers like BiCG-stab and QMR. The experiments are run on Tesla K20Xm hosted by an Intel Core i7 CPU. Performance Flops of SpMV using GHD with IKEM is $3\times$ that of using CuSPRSE for CSR or BSR storages and $1.25\times$ for HYB or DIA storages, respectively. The number of structured grid cells is varied between 8×10^3 and 2.6×10^5 and each cell has 2, 4, or 8 state variables. Similarly, the performance Flops of BiCG-stab solver using GHD with IKEM is $2.6\times$ that of BiCG-stab for CSR or BSR storages and $1.27\times$ for HYB or DIA storages with CuSPRSE and CuBLAS library calls. Results show significant performance improvements in SpMV and BiCG-Stab solver in response to proposed optimizations compared to other proposed implementations found in the literature using standard sparse storages and numerical library calls involving multiple-kernel invocations.

This work contributes towards the three features of the ISL described in Sect. 4—Intelligent Data Structure Interface (feature 1) and synthesizing custom storage scheme (feature 3) and optimizing the solver algorithm (feature 4).

5.2 Invocation of GPU device routines from OpenACC

Many Numerical Libraries (NLs) have been developed to help scientists porting common scientific methods into GPU devices. Automated parallelization is needed to make NLs more accessible from high-level languages (HLLs). Interoperability between automated parallelization technologies is still underrated by researchers in terms of easiness of use and performance. For example, calling CuBLAS GPU interface from a GPU CUDA code region requires the use of handles to enable concurrency, which is similar to calling from a CPU code region. To reduce the overhead, there is need to eliminate the handle use from inside an HLL and introduce an intermediate function that does not require cuBLASHandle argument. Hence, an intermediate implementation has been proposed that puts the necessary calls to create a new cuBLASHandle, call the actual library using that handle instance then destroy it after the call. This way the client code needs only to call the intermediate function without any handling of setup and destroy code for cuBLASHandles. A handle usually has a consistent pattern among each library. In the example of CuBLAS library; handle usage is fairly consistent among functions in a way that makes it possible to generalize a solution and automate it. Our approach is applicable to libraries that depend on pointers to opaque data structure handle arguments without a corresponding support from the compiler. A solution [3] for generating a wrapper library is proposed that avoids the need for such arguments; because these type of arguments stand as obstacles to such use. In the evaluation, the proposed solution showcase with a library paired with an OpenACC compiler that does not support it as an example. Our tests show speedups that reach 2.5× in some cases over the plain use of CuBLAS host-based interface, while the speedup reached about 34× with respect to the purely OpenACC-exclusive solution in some cases. Moreover, a decrease in code size of about 50% with respect to OpenACC-exclusive approach was noted..

This work facilitates the achievement of ISL's feature 2 "Automatic assessment and selection of a library storage scheme". Also it maps indirectly to ISL's feature 7 "Backward Compatibility and Extendibility".

5.3 Restructuring tool with auto-tuning

The design of a restructuring tool (RT-CUDA) [25] based on a restructuring algorithm is capable to convert a standard C-program (input) into an optimized CUDA program (output). The proposed restructuring algorithm acquires the best possible kernel optimizations and energy-aware rules. 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 software. 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 integrates both BA and

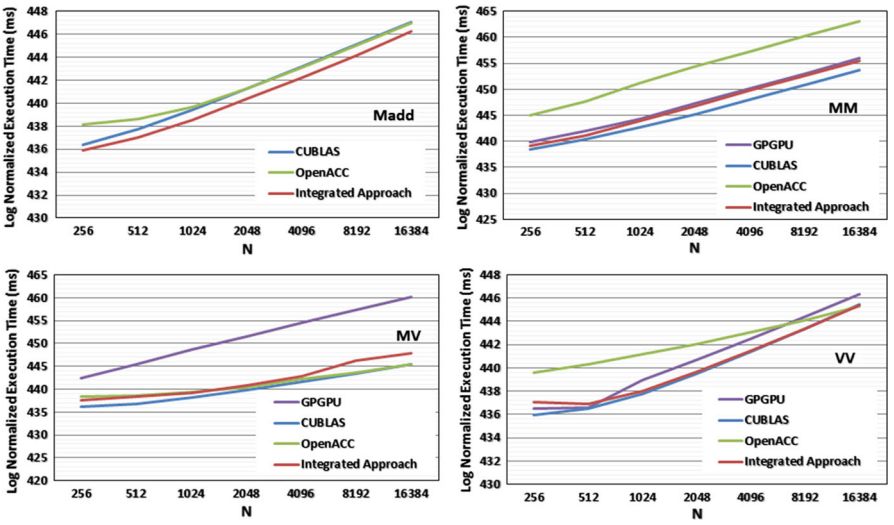

Fig. 2 Tools comparison using LAPACK operators

DS optimizations for code transformations. The user-defined functions generated by RT-CUDA are restructured as parametric CUDA kernels that are then pass through an efficient auto-tuning mechanism to enhance the GPU resource utilization by the functions.

A subset of the restructuring tools were evaluated with various applications including Matrix Addition (Madd), Matrix Multiplication (MM), Matrix–Vector Multiplication (MV), Vector-Vector Multiplication (VV), and also the recursive block matrix multiplication that are Strassen (S-MM) and Winograd (W-MM) Matrix Multiplications [24]. The evaluation results show the significant improvement in terms of the execution time of the parallel codes using the proposed integrated approach. The comparison for different applications and tools has shown with appropriate space size (N) and normalized execution time to show the results in a particular range.

Figure 2 shows the normalized execution times in milliseconds of different tools using a set of operators in LAPACK benchmark suite for basic linear algebra operations including Madd, MM, MV, and VV. The results show that our integrated approach obtained better performance for Madd and VV in comparison to CUBLAS library functions. However, for complex applications such as MM and MV, CUBLAS still has a significant performance advantage over our integrated approach. 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, the paper is focusing on high-level CUDA kernel optimizations. However, with the proposed high-level kernel optimizations, our integrated approach outperforms CUBLAS with 45% improvement in case of Madd and 2% improvement in case of VV. In addition to that, our integrated approach outperforms GPGPU compiler with 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

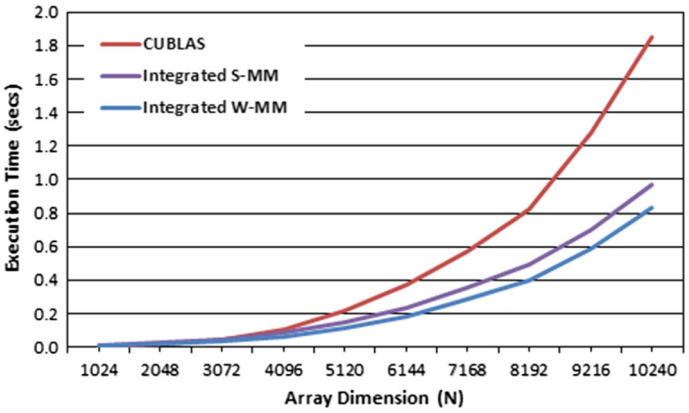

Fig. 3 Integrated approach for recursive block matrix multiplication

while our integrated approach generates correct values with any space size. Moreover, our integrated approach outperforms OpenACC implementation of PGI compiler with 42% improvement in case of Madd, 99% improvement in case of MM, and approx. similar performance in case of MV and VV for large arrays.

Using integrated approach, a recursive block matrix multiplication algorithms based on Strassen (S-MM) and its Winograd (W-MM) variant were also implemented. The above algorithms reduce the complexity of the canonical MM algorithm from O(N3) to O(N2.8). 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. Our implementation uses a small set of basic algebra functions, invoking CUBLAS, and use of auto-tuning of the parametric kernel to improve resource occupancy [24]. Evaluation (Fig. 3) shows that our 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 integrated S-MM and W-MM implementations with a few recursion levels can be used to further optimize the performance of basic algebra libraries. This work contributes towards the two ISL features—Optimizing the solver algorithm (feature 4) and Integrated auto-tuning (feature 5).

5.4 Conclusion

Billions of running threads are expected in the coming Exascale computing era. However, there is a mismatch between rapidly growing computational power and the efficiency of program produced by the current compilers and optimization tools. The scalable computing power of GPUs is highly essential for scientific simulations, especially for the class of Structured Grid Computing (SGC). In order to achieve higher simulation accuracy, large scale simulations of the problem with highly efficient code is required. These requirements are missing in the current technologies. This paper

identifies GPU architectural optimizations and techniques used in research and industry compilers to produce optimized code. It also identifies missing optimizations for the efficient implementation of SGC algorithms such as the iterative linear algebra solvers. Optimizing SGCs is found to be complex, error prone, and involve a variety of heterogeneous tools and techniques, which can be envisioned only from a research perspective. However, spreading the use of SGC on GPUs requires a deliberate effort for identifying the required automatic techniques for alleviating the complexity and the integration within a well-engineered framework. This paper details these techniques and described an integrated library with the required essential functionalities to ease the process of developing efficient storage, optimized code by using a high-level interactive interface and intelligent domain specific annotations.

Acknowledgements The authors would like to acknowledge the support provided by King Abdulaziz City for Science and Technology (KACST) through the Science and Technology Unit at King Fahd University of Petroleum and Minerals (KFUPM) for funding this work through project No. 12-INF3008-04 as part of the National Science, Technology and Innovation Plan.

References

1. Al-Mouhamed MA, Khan AH (2017) SpMV and BiCG-Stab optimization for a class of hepta-diagonal-sparse matrices on GPU. J Supercomput 73(9):3761–3795. https://doi.org/10.1007/s11227-017-1972-3
2. Aldinucci M, Danelutto M, Drocco M, Kilpatrick P, Misale C, Peretti Pezzi G, Torquati M (2018) A parallel pattern for iterative stencil + reduce. J Supercomput 74(11):5690–5705. https://doi.org/10.1007/s11227-016-1871-z
3. Almousa A (2017) Experimental evaluation and enhancement of optimizations of annotation-based and automatic parallel code generators for GPUs. PhD thesis, King Fahd University of Petroleum and Minerals
4. Alnæs MS, Logg A, Ølgaard KB, Rognes ME, Wells GN (2014) Unified form language: a domain-specific language for weak formulations of partial differential equations. ACM Trans Math Softw (TOMS) 40(2):9
5. Ansel J, Kamil S, Veeramachaneni K, Ragan-Kelley J, Bosboom J, O'Reilly UM, Amarasinghe S (2014) Opentuner: an extensible framework for program autotuning. In: Proceedings of the 23rd international conference on parallel architectures and compilation. ACM, pp 303–316
6. Anzt H, Tomov S, Luszczek P, Sawyer W, Dongarra J (2015) Acceleration of GPU-based krylov solvers via data transfer reduction. Int J High Perform Comput Appl 29(3):366–383
7. Bell N, Garland M (2009) Implementing sparse matrix–vector multiplication on throughput-oriented processors. In: Proceedings of the conference on high performance computing networking, storage and analysis. ACM, p 18
8. Beyer JC, Stotzer EJ, Hart A, de Supinski BR (2011) OpenMP for accelerators. In: IWOMP, lecture notes in computer science. Springer, pp 108–121
9. Bodin F, Bihan S (2009) Heterogeneous multicore parallel programming for graphics processing units. Sci Program 17(4):325–336
10. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (2008) A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not 43(6):101–113. https://doi.org/10.1145/1379022.1375595
11. Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P (2004) Brook for GPUs: stream computing on graphics hardware. ACM Trans Graph 23(3):777–786. https://doi.org/10.1145/1015706.1015800
12. Cevahir A, Nukada A, Matsuoka S (2010) High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning. Comput Sci Res Dev 25(1–2):83–91
13. Dagum L, Menon R (1998) OpenMP: an industry-standard API for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55. https://doi.org/10.1109/99.660313

14. Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson D, Shalf J, Yelick K (2008) Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE Press, p 4
15. Ernstsson A, Li L, Kessler C (2018) Skepu 2: flexible and type-safe skeleton programming for heterogeneous parallel systems. Int J Parallel Program 46(1):62–80. https://doi.org/10.1007/s10766-017-0490-5
16. Galvez R, van Anders G (2011) Accelerating the solution of families of shifted linear systems with cuda. arXiv:1102.2143
17. Gao J, Qi P, He G (2016) Efficient CSR-based sparse matrix-vector multiplication on GPU. Math Probl Eng. https://doi.org/10.1155/2016/4596943
18. Gebali F (2011) Algorithms and parallel computing, vol 84. Wiley, Hoboken
19. Godwin J, Holewinski J, Sadayappan P (2012) High-performance sparse matrix–vector multiplication on GPUs for structured grid computations. In: Proceedings of the 5th annual workshop on general purpose processing with graphics processing units. ACM, pp 47–56
20. Han TD, Abdelrahman TS (2011) hiCUDA: high-level GPGPU programming. IEEE Trans Parallel Distrib Syst 22:78–90. https://doi.org/10.1109/TPDS.2010.62
21. Huan G, Qian Z (2012) A new method of sparse matrix–vector multiplication on GPU. In: 2012 2nd International conference on computer science and network technology (ICCSNT). IEEE, pp 954–958
22. Kamil S (2009) A generalized framework for auto-tuning stencil computations. Lawrence Berkeley National Laboratory, Berkeley
23. Kamil S, Chan C, Oliker L, Shalf J, Williams S (2010) An auto-tuning framework for parallel multicore stencil computations. In: 2010 IEEE international symposium on parallel and distributed processing (IPDPS). IEEE, pp 1–12
24. Khan A, Al-Mouhamed M, Fatayer A, Mohammad N (2016) Optimizing the matrix multiplication using strassen and winograd algorithms with limited recursions on many-core. Int J Parallel Program 44(4):801–830
25. Khan AH, Al-Mouhamed M, Al-Mulhem M, Ahmed AF (2017) RT-CUDA: a software tool for CUDA code restructuring. Int J Parallel Program 45(3):551–594
26. Khan M, Basu P, Rudy G, Hall M, Chen C, Chame J (2013) A script-based autotuning compiler system to generate high-performance cuda code. ACM Trans Archit Code Optim 9(4):31:1–31:25. https://doi.org/10.1145/2400682.2400690
27. Lee S, Eigenmann R (2013) OpenMPC: extended OpenMP for efficient programming and tuning on GPUs. Int J Comput Sci Eng (IJCSE) 8(1):4–20
28. Leung A, Vasilache N, Meister B, Baskaran M, Wohlford D, Bastoul C, Lethin R (2010) A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In: Proceedings of the 3rd workshop on general-purpose computation on graphics processing units, GPGPU '10. ACM, New York, NY, USA, pp 51–61
29. Liao SW, Du Z, Wu G, Lueh GY (2006) 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
30. Lowell D, Godwin J, Holewinski J, Karthik D, Choudary C, Mametjanov A, Norris B, Sabin G, Sadayappan P, Sarich J (2013) Stencil-aware GPU optimization of iterative solvers. SIAM J Sci Comput 35(5):S209–S228
31. Mametjanov A, Lowell D, Ma CC, Norris B (2012) Autotuning stencil-based computations on GPUs. In: 2012 IEEE international conference on cluster computing (CLUSTER). IEEE, pp 266–274
32. Maruyama N, Aoki T (2014) Optimizing stencil computations for NVIDIA Kepler GPUs. In: Proceedings of the 1st international workshop on high-performance stencil computations, Vienna. pp 89–95
33. Mueller K, Xu F, Neophytou N (2007) Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography? Proc SPIE 6498:64980N – 6498 – 12
34. OpenMP: The OpenMP®API specification for parallel programming (2018). http://openmp.org/wp/. Accessed Jan 2019
35. Peercy M, Segal M, Gerstmann D (2006) A performance-oriented data parallel virtual machine for GPUs. In: SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches. ACM, New York, NY, USA, p 184
36. PGI: Portland group (2019). http://www.pgroup.com/resources/accel.htm. Accessed Jan 2019
37. Rivera-Polanco D (2009) Collective communication and barrier synchronization on NVIDIA CUDA GPU. Ms thesis, University of Kentucky

38. Sedaghati N, Ashari A, Pouchet LN, Parthasarathy S, Sadayappan P (2015) Characterizing dataset dependence for sparse matrix–vector multiplication on GPUs. In: Proceedings of the 2nd workshop on parallel programming for analytics applications. ACM, pp 17–24
39. Sedaghati N, Mu T, Pouchet LN, Parthasarathy S, Sadayappan P (2015) Automatic selection of sparse matrix representation on GPUs. In: Proceedings of the 29th ACM on international conference on supercomputing. ACM, pp 99–108
40. Tojo N, Tanabe K, Matsuzaki H (2014) US Patent and Trademark Office, Washington, DC, US Patent No. 8,732,684
41. Tomov S, Dongarra J, Baboulin M (2010) Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput 36(5):232–240
42. Ueng SZ, Lathara M, Baghsorkhi SS, Hwu WMW (2008) Cuda-lite: reducing GPU programming complexity. In: Amaral JN (ed) Languages and compilers for parallel computing. Springer, Berlin, pp 1–15
43. Volkov V, Demmel J (2008) Benchmarking GPUs to tune dense linear algebra. In: Proceedings of the ACM/IEEE conference on high performance computing. p 31
44. Wang Z, Xu X, Zhao W, Zhang Y, He S (2010) Optimizing sparse matrix–vector multiplication on CUDA. In: International conference on education technology and computer. https://doi.org/10.1109/ICETC.2010.5529724
45. Wikipedia: Algorithmic skeleton (2019). https://en.wikipedia.org/wiki/Algorithmic_skeleton. Accessed 01 June 2019
46. Williams S, Oliker L, Vuduc R, Shalf J, Yelick K, Demmel J (2009) Optimization of sparse matrix–vector multiplication on emerging multicore platforms. Parallel Comput 35(3):178–194
47. Xiao S, chun Feng W (2010) Inter-block GPU communication via fast barrier synchronization. In: IPDPS. pp 1–12
48. Yang M, Sun C, Li Z, Cao D (2012) An improved sparse matrix–vector multiplication kernel for solving modified equation in large scale power flow calculation on CUDA. In: IEEE 7th international power electronics and motion control conference—ECCE Asia