20th AIAA Computational Fluid Dynamics Conference
27 - 30 June 2011, Honolulu, Hawaii

AIAA 2011-3221

GPGPU parallel algorithms for structured-grid CFD codes

Christopher P. Stone[*]
Computational Science and Engineering, LLC, Seattle, WA, 98144, USA

Earl P.N. Duque[†]
Intelligent Light, Rutherford, NJ, 07070, USA

Yao Zhang[‡] David Car[§] John D. Owens[¶] and Roger L. Davis[‖]
University of California, Davis, Davis, CA, 95616, USA

A new high-performance general-purpose graphics processing unit (GPGPU) computational fluid dynamics (CFD) library is introduced for use with structured-grid CFD algorithms. A novel set of parallel tridiagonal matrix solvers, implemented in CUDA, is included for use with structured-grid CFD algorithms. The solver library supports both scalar and block-tridiagonal matrices suitable for approximate factorization (AF) schemes. The computational routines are designed for both GPU-based CFD codes or as a GPU accelerator for CPU-based algorithms. Additionally, the library includes, among others, a collection of finite-volume calculation routines for computing local and global stable time-steps, inviscid surface fluxes, and face/node/cell-centered interpolation on generalized 3D, multi-block structured grids. GPU block tridiagonal benchmarks showed a speed-up of 3.6x compared to an OpenMP CPU Thomas Algorithm results when host-device data transfers are removed. Detailed analysis shows that a structure-of-arrays (SOA) matrix storage format versus an array-of-structures (AOS) format on the GPU improved the parallel block-tridiagonal performance by a factor of 2.6x for the parallel cyclic reduction (PCR) algorithm. The GPU block tridiagonal solver was also applied to the OVERFLOW-2 CFD code. Performance measurements using synchronous and asynchronous data transfers within the OVERFLOW-2 code showed poorer performance compared to the cache-optimized CPU Thomas Algorithm. The poor performance was attributed to the significant cost of the rank-5 sub-matrix and sub-vector host-device data transfers and the matrix format conversion. The finite-volume maximum time-step and inviscid flux kernels were benchmarked within the MBFLO3 CFD code and showed speed-ups, including the cost of host-device memory transfers, ranging from 3.2–4.3x compared to optimized CPU code. It was determined, however, that GPU acceleration could be increased to 21x over a single CPU core if host-device data transfers could be eliminated or significantly reduced.

I. Introduction

Recently, programmable graphics processing units (GPUs) have received much attention due to their high floating-point rate and low power consumption compared to commodity multi-core CPUs. In fact, at the time of this writing, 3 of the top 5 fastest[1] and 10 of 20 most efficient[2] supercomputers in the world employ GPU-based accelerators. The GPU has historically been optimized for graphics operations; however, the relatively

[*]Computational Science and Engineering LLC, 1911 15th Ave South, Seattle, WA, 98144; AIAA Member; chris.stone@computational-science.com

[†]Manager, Applied Research Group, Intelligent Light, 301 Rt 17N, 7th Floor, Rutherford, NJ 07070; AIAA Senior Member; epd@ilight.com

[‡]Graduate Research assistant, Electrical and Computer Engineering, University of California, Davis; yaozhang@ucdavis.edu

[§]Graduate Student, University of California, Davis; AIAA member

[¶]Associate Professor, Electrical and Computer Engineering, University of California, Davis; jowens@ece.ucdavis.edu

[‖]Professor, Mechanical and Aerospace Engineering, University of California, Davis; AIAA Associate Fellow; roger-davis@ucdavis.edu recent addition of programmability to the graphics pipeline has allowed a variety of application domains to realize significant performance gains. These gains come from the combination of fine-grain parallelism, high arithmetic rates and high memory bandwidth compared to modern multi-core CPUs. Algorithms must be designed with a high degree of parallelism, specifically fine-grained parallelism, in order to exploit the available processing power of these GPUs.

The goal of this research endeavor is to develop and evaluate a set of commonly employed computational tools, used widely in modern structured-grid CFD codes, that can be efficiently implemented in CUDA. This collection of efficient yet generalized CUDA kernels can simplify the implementation of GPU acceleration in existing CFD codes. Several factors critical to efficient GPU performance were addressed during the development such as memory layout, data transfer costs and parallelism. Since many structured-grid CFD algorithms share a similar design, these kernels can also act as templates for further GPU development.

Two classes of CUDA-based GPGPU utilities have been developed and will be introduced in the following sections: (i) parallel block-tridiagonal solvers for alternating direction implicit (ADI) CFD algorithms and (ii) a set of parallel finite-volume utilities for computing grid reduction operations (e.g., min, max, sum, etc.); local and global stable time-step size; inviscid (i.e., Euler) fluxes; and numerical dissipation schemes for multi-block, explicit algorithms. The parallel block tridiagonal solvers are the first known implementation of their kind in CUDA.

In the following section, several parallel scalar and block tridiagonal solution algorithms are introduced and their performance characteristics are analyzed as they pertain to GPU implementations. Benchmark results are then given for each of these algorithms. The first benchmark uses a simplified and easily controlled test problem. The parallel algorithms are then benchmarked within the *approximate factorization* Beam and Warming[3] scheme of the OVERFLOW-2.[4] OVERFLOW-2 is a widely used overset-grid compressible flow solver from NASA. The second class of GPGPU CFD utilities, the finite-volume time-step and inviscid flux calculators, are then implemented and benchmarked within the 3d, multi-block CFD code MBFLO3.[5,6] All GPU benchmarks presented were performed on the latest NVIDIA computational hardware (e.g., *Fermi* C2050) and using single and double-precision arithmetic. Comparisons with optimized, parallel CPU-based algorithms are presented to give real-world speed-up characteristics.

II. Tridiagonal Methods

Tridiagonal matrix systems arise frequently in finite-difference approximations for ordinary differential equations in a wide range of applications. Examples abound, including the finite-difference approximations for the 1-d Poisson and Laplace equations and spectral methods for the 2-d Poisson equation. As as result, the rapid solution of the tridiagonal systems is critical for efficient numerical solutions.

A typical asymmetric tridiagonal matrix is shown in Eqn. 1. There, a, b and c represent the lower, middle, and upper diagonal bands and u and f the solution and right-hand-side (RHS) vectors.

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \dots & 0 & a_n & b_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix} \tag{1}$$

The diagonal and RHS vector elements may be scalars or, more generally, sub-matrices and sub-vectors. In this work, tridiagonal matrices of scalar elements shall be referred to as *scalar* tridiagonal systems and those with sub-matrix elements as *block* tridiagonal systems.

The *approximate factorization* (AF) and *alternating direction implicit* (ADI) methods[7] are often used in computational fluid dynamics (CFD) solutions of the Navier-Stokes equations (NSE), the governing equations for fluid flow. These methods are designed to implicitly solve the NSE on structured grids using finite-differences. AF schemes factorize a multidimensional problem into a sequence of simpler and, more importantly, cheaper one-dimensional problems. The result of the factorization is a set of block tridiagonal matrix systems. Another tridiagonal CFD application is high-order compact (HOC) finite-difference schemes.[8] There, a variety of implicit finite-difference equations for the spatial derivatives result in scalar tridiagonal systems. These schemes are popular due to their spectral-like accuracy while retaining a compact stencil on structured grids. The rapid and accurate inversion of these systems is critical for an efficient CFD algorithm.

In the AF and ADI schemes, the tridiagonal systems along a coordinate direction are independent and may be processed concurrently resulting in a coarse level of parallelism. The coarse-grained approach is well-suited for many vector and multi-processor systems. However, the granularity of multi-core systems is rapidly increasing requiring algorithms that can exploit finer levels of parallelism. For example, modern programmable graphical processing units (GPUs) are becoming available that contain many hundreds of processing cores.

The solution of Eqn. 1 is commonly found through the Thomas algorithm (TA). TA is a specialization of Guassian Elimination that exploits the tridiagonal structure of the system and has a computational time $O(nm^3)$ where m is the rank of the elements and n is the system size (i.e., number of rows). While the algorithm operates in linear time with n, it exhibits only $O(1)$ parallelism. As such, it is poorly suited for the fine-grained GPU parallel environment.

The parallel solution of Eqn. 1 has been investigated for many decades. Hockney and Jesshope[9] gave a detailed numerical and performance analysis of the Cyclic Reduction (CR) algorithm for scalar systems on theoretical pipeline and distributed memory parallel systems. More recently, Hirshman et al.[10] presented a detailed performance analysis and benchmark study of a block tridiagonal CR solver, BCYCLIC, on a modern distributed memory cluster with multiple cores per node. There, the sub-matrix element rank (m) was assumed to be large (i.e., $m \gg 1$) that limits the relevance to the intended CFD setting. In AF schemes, sub-matrix elements are typically 5x5 (i.e., rank-5) for three-dimensional problems. However, their generalized performance analysis highlights the salient features of the TA and CR algorithms for block tridiagonal systems.

Zhang et al.,[11] Davidson et al.,[12] Davidson and Owens[13] and Goddeke and Strzodka[14] reported on the performance of CR applied to scalar tridiagonal system on GPUs. In Zhang et al.,[11] CR was compared to Parallel Cyclic Reduction (PCR), Recursive Doubling (RD) and hybrid combinations of the three. These were all compared to dense solver solutions showing a factor of up to 12 speed-up. They found that the combination of CR and PCR gave the best overall performance (speed and stability) for scalar systems. They also noted that the various algorithms were highly sensitive to shared memory bank conflicts in the NVIDIA GPUs using CUDA. Goddeke and Strzokda,[14] however, did not observe the same issue in their CR performance studies. There, the tridiagonal solver was used as a smoothing algorithm within a broader mixed-precision, multigrid Poisson solver, an application with many similarities to the ADI scheme. Davidson et al.[12] introduced a method for efficiently solving arbitrarily large scalar tridiagonal systems on the GPU and Davidson and Owens[13] presented a *register packing* optimization method for improving the scalar tridiagonal CR solver. Sakharnykh[15] reported on the use of the Thomas Algorithm to concurrently solve multiple scalar tridiagonal systems on the GPU within an incompressible CFD ADI scheme. There, parallelism across many 100s or 1000s of concurrent systems is exploited, a straightforward strategy commonly employed in multi-core CPU systems. The current research focuses on the solution of block tridiagonal systems and studies the performance of CUDA-based parallel CR and PCR algorithms.

III. Tridiagonal Algorithms

Several algorithms for solving tridiagonal matrices are presented in this section. In each description, the tridiagonal matrix has n rows and the individual elements are dense rank-m sub-matrices. Further, n is assumed to a power-of-two (i.e., $n = 2^k$). This is not a requirement for the following algorithms but greatly simplifies the description and analysis. The extension of the algorithm to non-powers-of-2 is described in Hirshman et al.[10]

The Thomas algorithm (TA) is the most common solution method for tridiagonal systems. TA is a specialized application of Gaussian elimination taking into account the banded structure of the tridiagonal system. TA proceeds in two phases: forward elimination and backward substitution. In the forward elimination phase, the lower diagonal is eliminated from the second row onward. Backward substitution then begins from the nth row, which can be directly solved, and recursively solves each row in reverse order. The total computational work is $O(nm^3)$ and takes $2n - 1$ steps. However, both the forward and backward phases must proceed sequentially (or serially), resulting in only $O(1)$ parallelism.

During the TA forward elimination phase, the kth row's lower element (a_k) is sequentially eliminated using the following equations:

$$
\begin{aligned}
a'_k &= a_k - a_k b_{k-1}^{-1} b_{k-1} \\
b'_k &= b_k - a_k b_{k-1}^{-1} c_{k-1} \\
c'_k &= c_k \\
f'_k &= f_k - a_k b_{k-1}^{-1} f_{k-1}
\end{aligned} \tag{2}
$$

where a'_k is equivalently zero. As can be seen, the elimination of a_k is dependent upon a_{k-1} having been previously eliminated leading to the serial recursion.

A parallel alternative to TA is Cyclic Reduction (CR).[9] In CR, half of the rows are recursively eliminated at each reduction step. After $\log_2(n)$ steps, the system is reduced to a single equation that is then solved directly. Then, backward recursion allows the solution at each step to be found from the preceding step exclusively. At every forward and backward step, there is $O(n/2^p)$ parallelism where $p = 1$ to $\log_2(n)$. A potential weakness of this algorithm is the decrease in parallelism towards the end of the reduction phase: the parallelism tends to 1.

The CR method recursively eliminates the upper and lower diagonal coefficients with the following relations for each step p.

$$
\begin{aligned}
a'_k &= \quad - a_k b_{k-\delta}^{-1} a_{k-\delta} \\
b'_k &= b_k - a_k b_{k-\delta}^{-1} c_{k-\delta} - c_k b_{k+\delta}^{-1} a_{k+\delta} \\
c'_k &= \qquad\qquad\qquad\quad - c_k b_{k+\delta}^{-1} c_{k+\delta} \\
f'_k &= f_k - a_k b_{k-\delta}^{-1} f_{k-\delta} - c_k b_{k+\delta}^{-1} f_{k+\delta}
\end{aligned} \tag{3}
$$

where $\delta = 2^{p-1}$. Note, since only the even rows are modified at each step in CR, the coefficients can be updated in-place (i.e., no temporary storage is required). TA has this same feature. The total storage requirement for CR and TA is $O(n(3m^2 + 2m))$.

Parallel Cyclic Reduction (PCR) is a variant of CR that is designed to maintain uniform parallelism throughout the reduction steps. The same reduction scheme is applied (i.e., Eqn. 3) in PCR as in CR. However, instead of only eliminating the even rows at each stage, both the even *and* odd rows are eliminated in PCR. At each reduction step, the current system is split into two linearly independent systems: one for the even and one of the odd rows. After $p = \log_2(n)$ forward reduction steps, the original matrix system is reduced to n linear independent equations each that can be directly solved. As a result, PCR does not require a reserve solution phase. And, as designed, the parallelism of PCR is maintained at n throughout the algorithm.

Since PCR eliminates both the even and odd equations at each step, the updated coefficients must be temporarily stored. The total storage requirement for PCR scales as $O(n(5m^2 + 2m))$, roughly 66% greater than CR assuming $m \gg 1$.

III.A. Computational effort

A detailed listing of the computational complexity and parallelism for each algorithm is given in Table 1. The scaling estimates are for a block tridiagonal matrix with n rows and sub-matrix elements of rank m. The columns from left to right present the number of steps, the total computational work per step, the total integrated work, the available parallelism and the parallel time assuming perfect parallel machine. Estimates are given for the forward and reverse phases from top to bottom for each algorithm. The symbols C_{inv}, C_{mat}, and C_{vec} represent the cost of matrix inversion, matrix-matrix and matrix-vector multiplications, respectively, for a given platform and implementation. (For this analysis, only the scaling and not the distinct costs of each are relevant.) The cost of these matrix and vector operations scale as m^3, m^3 and m^2, respectively. In practice, LU factorization[10] is used instead of direct inversion but does not affect the original m^3 scaling.

Comparing the above estimates, TA is seen to have the lowest total work of the three. CR has the same scaling as TA but requires relatively more work in total. Hirshman et al.[10] estimated the CR-to-TA work ratio as approximately 2.6. As a result, it can be assumed that a minimum parallelism of three is required by CR to be match the TA serial performance. With hundreds of processing cores per device, the intended

Table 1. Comparison of Thomas (TA), Cyclic Reduction (CR) and Parallel Cyclic Reduction (PCR) algorithms applied to a block tridiagonal matrix with n rows and with rank-m sub-matrix elements.

Algorithm	Number of steps	Work per step (k)	Total work	Parallelism per step (k)	Parallel time
TA	n	$C_{\text{inv}} + 2C_{\text{mat}} + C_{\text{vec}}$	nm^3	1	nm^3
	$n-1$	C_{vec}	nm^2	1	nm^2
CR	$\log_2 n$	$n/2^k(C_{\text{inv}} + 6C_{\text{mat}} + 2C_{\text{vec}})$	nm^3	$n/2^k$	$m^3 \log_2 n$
	$\log_2 n - 1$	$n/2^k(2C_{\text{vec}})$	nm^2	$n/2^k$	$m^2 \log_2 n$
PCR	$\log_2 n$	$n(C_{\text{inv}} + 6C_{\text{mat}} + 2C_{\text{vec}})$	$nm^3 \log_2 n$	n	$m^3 \log_2 n$
	1	$n(C_{\text{inv}} + C_{\text{vec}})$	nm^3	n	m^3

GPU devices should far exceed this limit. Unlike CR and TA, the cost of PCR is not linear with n but increases by a factor of $\log_2 n$. That is, the total computational work for PCR increases with the system size and may negate the improved parallelism for large n.

It is important to note that the above algorithms can be applied equally to scalar and block tridiagonal systems. Additional arithmetic optimizations are possible for scalar elements (e.g., matrix inversion is replaced with simple scalar division). However, these differences do not affect the theoretical performance analysis presented above.

III.B. CFD Application: Approximate Factorization Scheme

The governing fluid dynamics equations, the Navier-Stokes equations, can be written in strong conservative form and in a generalized 3-d coordinate system as,

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{E}}{\partial \xi} + \frac{\partial \mathbf{F}}{\partial \eta} + \frac{\partial \mathbf{G}}{\partial \zeta} = 0 \quad \ldots \quad \mathbf{U} = \frac{1}{J}[\rho, \rho u, \rho v, \rho w, \rho e_0]^T \tag{4}$$

where ρ, (ρu, ρv, ρw) and ρe_0 are the density, momentum and stagnation energy. The Beam and Warming[3] (BW) implicit time-step scheme is widely used to solve Eqn. 4 on structured grids. In the BW algorithm, the nonlinear flux vectors ($\mathbf{E}$, $\mathbf{F}$, $\mathbf{G}$) are linearized about the current time-step to form a linear implicit scheme. For example,

$$\mathbf{E}^{n+1} \approx \mathbf{E}^n + [A](\mathbf{U}^{n+1} - \mathbf{U}^n) \tag{5}$$

where $[A]$ is the flux Jacobian matrix (e.g., $[A] = \partial \mathbf{E}/\partial \mathbf{U}$). The flux Jacobian matrices in the η and ζ directions are similarly defined as $[B] = \partial \mathbf{F}/\partial \mathbf{U}$ and $[C] = \partial \mathbf{G}/\partial \mathbf{U}$.

After linearization, an implicit scheme can be written compactly with $\mathbf{U}^{n+1} = \mathbf{U}^n + \Delta \mathbf{U}^{n+1}$ as

$$\left\{ I + \frac{\delta t}{1+\phi} \left(\frac{\partial [A]}{\partial \xi} + \frac{\partial [B]}{\partial \eta} + \frac{\partial [C]}{\partial \zeta} \right) \right\} \Delta \mathbf{U}^{n+1} = \frac{\phi}{1+\phi} \Delta \mathbf{U}^n - \frac{\delta t}{1+\phi} \left\{ \frac{\partial \mathbf{E}}{\partial \xi} + \frac{\partial \mathbf{F}}{\partial \eta} + \frac{\partial \mathbf{G}}{\partial \zeta} \right\}^n . \tag{6}$$

The temporal accuracy can be varied from 1^{st} to 2^{nd}-order by adjusting the free parameter ϕ. Using second-order central differences for the LHS derivatives leads to a banded block matrix with three upper and lower-diagonals of length $N_\xi \times N_\eta \times N_\zeta$ and where each element is a 5x5 sub-matrix. Nichols et al.[16] solved Eqn. 6 iteratively using a successive symmetric over relaxation (SSOR) scheme. Alternately, Eqn. 6 can be *approximately factorized* (AF) into three components as

$$\left\{ I + \frac{\delta t}{1+\phi} \left(\frac{\partial [A]}{\partial \xi} \right) \right\} \left\{ I + \frac{\delta t}{1+\phi} \left(\frac{\partial [B]}{\partial \eta} \right) \right\} \left\{ I + \frac{\delta t}{1+\phi} \left(\frac{\partial [C]}{\partial \zeta} \right) \right\} \Delta \mathbf{U}^{n+1} = \mathbf{R}^n \tag{7}$$

Here, $\mathbf{R}$ is the right-hand side (RHS) of Eqn. 6. Note, Eqn. 7 differs from the original by terms on the order of δt^2 and δt^3. This error tends to zero at convergence and does not significantly impact the formal accuracy of the scheme ($O(\delta t^2$,$\delta x^2)$) in steady-state problems.

Equation 7 is solved in three sequential steps:

$$\begin{aligned}
\left\{I + \frac{\delta t}{1+\phi}\left(\frac{\partial[A]}{\partial\xi}\right)\right\}\Delta\mathbf{U}^{\xi} &= \mathbf{R}^{n} \\
\left\{I + \frac{\delta t}{1+\phi}\left(\frac{\partial[B]}{\partial\eta}\right)\right\}\Delta\mathbf{U}^{\eta} &= \Delta\mathbf{U}^{\xi} \\
\left\{I + \frac{\delta t}{1+\phi}\left(\frac{\partial[C]}{\partial\zeta}\right)\right\}\Delta\mathbf{U}^{n+1} &= \Delta\mathbf{U}^{\eta}
\end{aligned} \tag{8}$$

where $\Delta\mathbf{U}^{\xi}$ and $\Delta\mathbf{U}^{\eta}$ are intermediate solutions. Each step requires the inversion of a set of one-dimensional block tridiagonal matrices whose length is equal to the coordinate dimension (e.g., N_{ξ}). The solution of these tridiagonal matrices is significantly cheaper than solving Eqn. 4 in both computational time and storage requirements.

While the approximately factorized BW scheme must proceed in sequence, several levels of parallelism still exist. For a given coordinate direction, inversion along the two remaining coordinates can proceed concurrently. For example, when solving along the ξ coordinate, there are $N_{\eta} \times N_{\zeta}$ independent tridiagonal matrices that can be solved in parallel. Further, the construction of the LHS block matrix coefficients at each step can occur concurrently facilitating functional pipelining. The combination of these two strategies is expected to fit well within the targeted throughput-oriented GPU devices. However, only the parallel solution of the block tridiagonal matrices is investigated presently.

III.C. CUDA Implementation

The current work employs NVIDIA's *Compute Unified Device Architecture* (CUDA). This architecture supports fine-grained data parallelism through a hierarchical multi-processor and multi-core design. These devices are designed around one or more streaming multiprocessors (SMs). Each SM in turn contains a set of scalar processors (SPs) along with local on-chip shared memory. All processors have access to the GPU global memory at high bandwidth. Threads are scheduled rapidly via a hardware-based thread workload manager on each SM. Blocks of threads are launched together and can work collectively through per-block synchronization instructions. Increased performance is achieved by scheduling multiple blocks per SM. This allows memory latency to be hidden increasing the realized throughput.

To achieve reasonable performance in CUDA, it is necessary to properly manage memory access. Unlike commodity CPUs, CUDA does only limited hardware-managed memory caching that can quickly cause execution kernels to become memory bound. Reads from global memory can be accelerated if all threads within a block access data in a contiguous and regularized fashion. Under specific conditions, reads and writes to global memory from a block can be *coalesced* as a single memory operation with a significant improvement in throughput. This mechanism is exploited where possible.

The parallel block tridiagonal algorithms described earlier were implemented in CUDA following the work of Zhang et al.[11] In their work, each matrix system is solved by a single thread block. CR uses $n/2$ threads per thread block while PCR uses n threads per thread block. This same thread strategy is used for the new block tridiagonal algorithms. The CR and PCR block tridiagonal matrix algorithms use a generalized rank-m LU decomposition method to invert the sub-matrix elements. The sub-matrix and sub-vector operations are done in-place within shared memory as much as possible. This is done to reduce register pressure and increase the possible occupancy rate. The scalar CR and PCR implementations employ traditional scalar division and multiplication.

As mentioned above each block has access to a fast on-chip shared memory segment. Before solving the tridiagonal systems, the matrix and vector bands are first read from the device global memory into shared memory by each block. The available shared memory, currently 48 kB per SM, limits the maximum supported matrix size. The largest possible rank-5 block tridiagonal matrix is 128 with CR when using single-precision. However, 128 is likely sufficient for many CFD applications. Davidson et al.[12] recently presented a method for solving arbitrarily large scalar tridiagonal systems on the GPU. In their work, PCR is recursively applied to a large scalar tridiagonal system effectively creating many smaller, independent tridiagonal systems. Future studies will examine their method as it applies to large block tridiagonal systems.

Before any calculations can occur on the GPU, the data must be transferred from the CPU host to the GPU device. This can commonly cause significant overhead that limits the realized performance. The impact of the transfer latency and bandwidth can be reduced by executing multiple kernels within *streams* that enables overlapped communication and computation.

IV. Results

The performance characteristics of the new block tridiagonal solvers and the finite-volume kernel toolkit are presented in this section. CPU performance is also presented for direct comparison.

IV.A. Block-tridiagonal performance

A series of benchmarks were conducted using the parallel tridiagonal algorithms discussed earlier. The tridiagonal matrix solvers are used to solve a finite-difference approximation for the 1-d Poisson equation.

$$\frac{d^2u}{dx^2} = e^x \qquad x = [0,1], u(0) = 0, u(1) = 1 \tag{9}$$

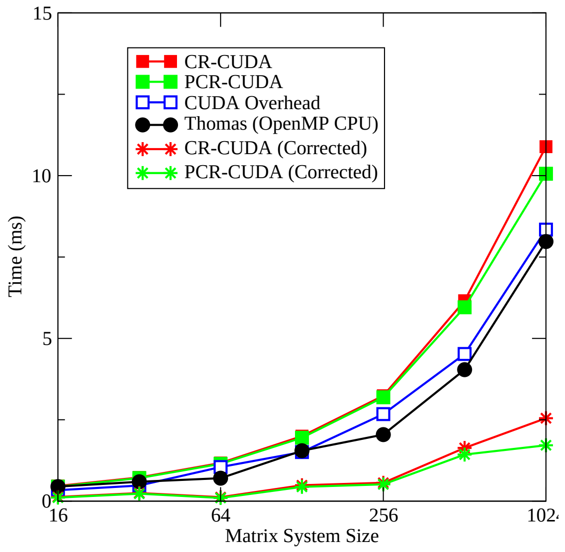
This type of problem was chosen since it has an analytical solution that can be used for error analysis and validation. A finite-difference approximation of Eqn. 9 is

$$u_{i-1} - 2u_i + u_{i+1} = h^2 f(x_i) \tag{10}$$

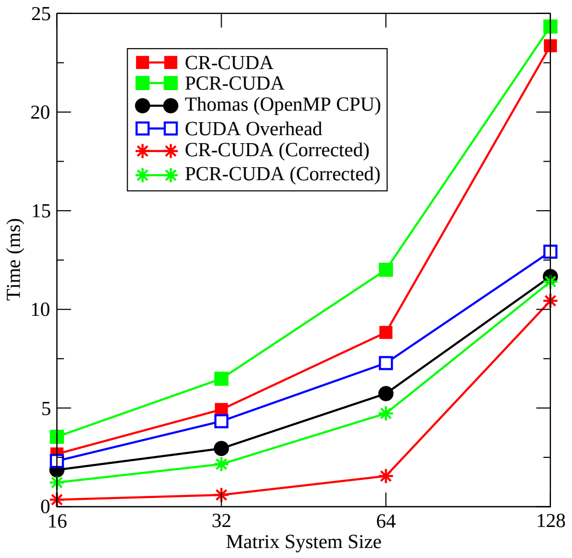
where $h = 1/(n+1)$ is the mesh spacing and n is the number of solution points. The resulting matrix system can be cast as a scalar or block tridiagonal matrix. The symmetric structure of the upper and lower diagonals is ignored as is the sparsity of the sub-matrix blocks.

The benchmarks were run on an AMD Opteron dual-processor system (Opteron 250, 2.4 GHz) with a single NVIDIA C2050 (Fermi) GPU. The host and device are linked via a PCI Express x16 graphics bus. OpenMP is used to parallelize the baseline TA results on the host. Each OpenMP thread solves a separate tridiagonal matrix system concurrently using TA.

Rank-1 (scalar) and rank-4 (block) matrix systems were investigated. For each test, the system length is varied from 16 to 1024 for rank-1 and 16 to 128 for rank-4. Again, the block matrix system size was restricted by the available shared memory. 512 copies of the test problem are solved concurrently for each test run and the benchmark results are averaged over 100 separate tests. All computations are done in single-precision (32-bit) arithmetic.



(a) rank-1



(b) rank-4

Figure 1. Benchmark results for rank-1 and rank-4 tridiagonal matrix systems using TA, CR and PCR algorithms. Vertical axis is total time (milliseconds) to solve 512 systems averaged over 100 samples. Open symbol denotes host-device communication time incurred during CR and PCR CUDA solvers. Stars denote CUDA solver times without communication overhead (i.e., corrected).

The timing results are shown in Fig. 1. The average total execution time is given as a function of the system size (n) for both the rank-1 and rank-4 problems using TA, CR and PCR. The total incurred overhead for the CUDA benchmarks is also shown. The overhead includes host-to-device and device-to-host data transfers and device memory allocation/deallocation. The *corrected* time shown for the parallel CUDA solvers is the solution run-time with the data transfer overhead removed.

As can be seen in Fig. 1, the CUDA CR and PCR algorithms do not result in performance improvements compared to the OpenMP TA algorithm using either rank-1 and rank-4 elements. In fact, the rank-4 performance is significantly slower than the OpenMP version.

Before computations are possible on the CUDA device, the block matrix data must be transferred from the CPU host to CUDA device through the PCI-Express bus. This transfer overhead is shown as a solid circle in Fig. 1 and is the same for both CR and PCR. For the scalar (rank-1) matrix systems, the overhead accounts for 70% of the total execution time and is approximately equal to the TA cost. The transfer overhead is less substantial for the block matrix (rank-4) case due to the higher computational intensity. For $n = 128$, the overhead consumes 48% of the execution time in CR. When the communication overhead is subtracted out, the scalar performance is significantly better than the OpenMP TA. The difference is less pronounced for the rank-4 block tridiagonal systems lowering the execution time roughly equal to the TA baseline results.

There is little difference between the CR and PCR algorithms for the scalar systems while CR outperforms PCR for rank-4. However, an interesting feature is observed for the rank-4 problem. As n is increased toward the 128 shared memory limit, the cost using CR is seen to increase at a greater rate than PCR. Two possible explanations are the decrease in parallelism and shared memory bank conflicts. As detailed earlier, CR's parallelism decreases with each reduction step. The low parallelism during the final few reduction steps could impact the realized performance. Memory bank conflicts can limit the shared memory bandwidth significantly impacting throughput.

The effect of shared memory bandwidth and bank conflicts can be examined by reordering the matrix/vector storage scheme. The tridiagonal sub-matrix and sub-vector elements can be written generically as a_{ijk} and f_{jk} where i and j denote the sub-element components and k denotes the tridiagonal matrix row. The elements can be stored in the *structure-of-arrays* (SOA) or *array-of-structures* (AOS) formats. In SOA, the individual components (e.g., $a_{11\,k}$) are stored contiguously for all rows k. AOS reverses this by storing all components for a single row k in order. AOS is the more natural format for an object-oriented method and is well-suited for cache-based or pipelined processors. However, SOA is more suited for vector processing (e.g., SSE and CUDA SIMT). The SOA format is expected to be more efficient for CUDA due to coalesced global memory access and reduced (or removed) shared memory bank conflicts.

The CUDA results in Fig. 1 employed SOA while the OpenMP TA results used AOS. The CUDA CR and PCR times using AOS are shown in Fig. 2. The SOA results from Fig. 1 are repeated for easy comparison. Note, only the rank-4 results are shown as the formats are equivalent for scalar systems. The performance impact of SOA compared to AOS is quite significant. The difference for $n = 128$ is approximately 2x for CR and 2.6x for PCR.

The benchmarks presented above show that the parallel solvers can provide performance gains compared to an optimized OpenMP Thomas Algorithm solver if the host-to-device transfer costs can be removed or significantly reduced. A 3–4x performance gain was realized using the PCR algorithm when the transfer cost is removed. We also show a strong dependence upon the data storage format on the CUDA device. Storing data in a structure-of-array (SOA) format resulted in a 2x performance gain compared to the array-of-structures (AOS) format.

The CR parallel block tridiagonal solver was also implemented within OVERFLOW-2 and benchmarked using the same hardware environment as before. OVERFLOW-2 uses the Beam and Warming[3] scheme to solve the compressible RANS equations on overset structured grids. OVERFLOW-2 applies the AF scheme (i.e., Eqn. 8) along coordinate planes to enable host-side vectorization. This algorithmic design will be exploited to allow asynchronous and concurrent execution of the parallel block-tridiagonal solvers. That is, multiple coordinate planes can be concurrently solved allowing an overlap of host-device data transfer and device kernel execution. The Intel Fortran compiler was used to compile the OVERFLOW-2 code using full optimization. Further, the cache-optimized version of the source-code was used to best fit the host's hardware.

For this benchmark, only the ξ-direction sub-step of the AF scheme was solved using the CUDA CR algorithm for simplicity. The remaining two sub-steps were solved using the internal cache-optimized

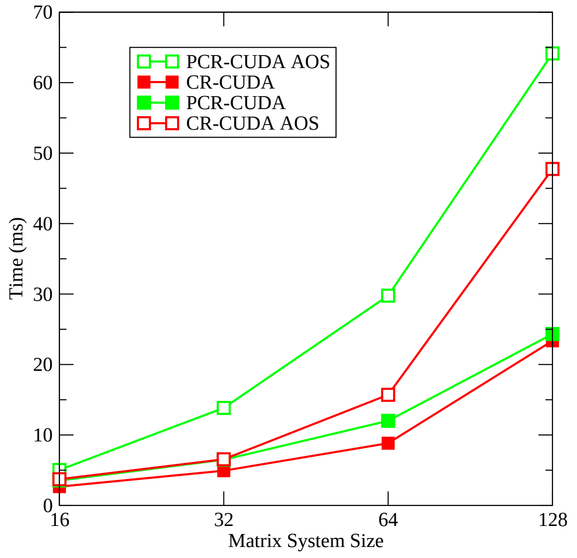


Figure 2. CUDA CR and PCR benchmark results comparing AOS and SOA data formats. Vertical axis is total time (milliseconds) to solve 512 systems averaged over 100 samples.

OVERFLOW-2 block tridiagonal solver. Note, all flux Jacobian matrices and residual vectors were computed on the CPU host; only the block tridiagonal matrix inversion was computed on the GPU. Therefore, the three 5x5 LHS sub-matrices and the 5x1 RHS sub-vector were transferred to the GPU before the matrix inversion began. The solution vector is copied from the GPU back to the host once the systems were solved.

A flat-plate boundary-layer simulation was used as a benchmark. A single grid was used for the current tests (i.e., no overset grid capabilities were required). As the prototype CR algorithm is restricted to tridiagonal systems of size 2^k, the tested grid sizes were 34, 66 and 130 points in all directions. (Two points were added in each direction to account for the specified boundary points.) N_η x N_ζ tridiagonal systems were solved during each ξ-direction sweep of the AF Beam-Warming scheme. This corresponds to 1024, 4096, and 16384 matrix systems of size 32, 64 and 128, respectively.

The cache-optimized ξ-direction routine in OVERFLOW-2 operates on a single computational coordinate line at a time. This is highly efficient for the host CPU hardware and uses the AOS storage format. As shown earlier, the SOA format is most efficient for the CUDA CR parallel algorithm. Therefore, the LHS sub-matrices and sub-vectors are copied and reformatted in temporary host memory before copying to the GPU device and vice-versa; this copy cost is not negligible. Only the most efficient formats for each platform are reported when comparing CPU and GPU performance results. That is, AOS is reported for the CPU and SOA is reported for the GPU. Note, a vector-optimized version is available in OVERFLOW-2 as well. While the resulting sub-matrix and sub-vector structure is more compatible with the CUDA CR algorithm, the time required to construct the matrix coefficients is significantly slower and is, therefore, not used.

Three different CUDA implementations were tested: (i) a single data transfer with a single kernel invocation; (ii) multiple asynchronous transfers with a single kernel; and (iii) multiple asynchronous transfers with multiple asynchronous kernels. In (i), all sub-matrix and sub-vector coefficients are computed at every grid point, converted to the SOA format and then transferred to the GPU in synchronized steps. Then, all the block tridiagonal systems are solved within a single CUDA kernel. The solution is finally transferred back to the host in a single transfer operation and the solution converted back to the native AOS format. Note, the host (CPU) storage requirement is a significantly higher for the CUDA implementations compared to the cache-based CPU version. In all three CUDA methods, storage for the coefficient matrices at each point on the grid is required whereas only a single ξ line of coefficients is stored at a one time in the cache-based CPU algorithm.

In (ii), the matrix coefficients are constructed by coordinate planes (i.e., ξ-η planes). After each ξ-η plane is populated and reformatted, the data is asynchronously transferred to the GPU. Asynchronous transfers are non-blocking and allow the CPU host thread to continue processing. In this case, the next ξ-η plane can be assembled while data is being transferred over the PCI bus. This is facilitated by CUDA *streams*, CUDA's asynchronous facility. After all ξ-η planes have been transferred, a single CUDA CR kernel is launched and bound to the transferring stream. The stream forces the kernel to block till all data transfers to the device have completed. As before, a single (blocking) device-to-host transfer copies the solution back to the host.

Finally, algorithm (iii) operates fully asynchronously. After each ξ-η plane is assembled by the OVERFLOW host code, the plane's data is asynchronously transferred to the device within a unique stream. A CUDA CR kernel and asynchronous device-to-host transfer is then bound to the unique stream. Each kernel waits till the host-to-device transfer is complete. Similarly, each device-to-host transfer waits till the stream's kernel completes execution. All of the asynchronous commands are non-blocking on the host allowing the CPU to proceed with the next ξ-η plane. This mechanism can reduce the overhead incurred by data transfers by overlapping host-device and device-host transfers with CPU calculations. However, some CPU resources are still consumed by the CUDA driver to facilitate the data transfer and kernel execution.

Performance measurements are shown in Table 2 for the three synchronization methods. Both single and double-precision results are given for each grid size. The CUDA block tridiagonal solvers were slower than the cache-optimized CPU solver. This is similar to the Poisson equation tests presented earlier. The synchronous method was slower by a factor of 2x for single-precision and 3x for double-precision. The data transfer accounted for approximately 29.6% of the total CUDA solver time for the single-precision fully-synchronized method. A significant improvement was observed when using the fully asynchronous method; however, it is still slower than the CPU method.

The current OVERFLOW-2 CUDA implementation was limited to solving only the block tridiagonal matrix systems. These results show no performance improvement, quite the opposite in fact, due to the overhead of data transfer and, to a lesser extent, reformatting. On a grid of size n^3, $240n^3$ words must be transferred from the host to GPU when solving the block tridiagonal matrices. Also, $15n^3$ words must be transferred back to the host, $5n^3$ for each of the three stages in the AF-BW scheme. However, solving the entire AF-BW scheme on the GPU requires the transfer of only $10n^3$ words to the device and $5n^3$ from the device; a factor of 17x difference in total. The amount of data transfer required is reduced since only the RHS residual flux vector and the conservative variables (i.e, the $\mathbf{U}$ in Eqn. 8), need to be copied to the device each time-step. The sub-matrix coefficients can be constructed directly on the GPU from the conservative variables. Computing these block sub-matrices accounts for approximately 46% of the floating point operations in the OVERFLOW-2 BW implementation. Building the sub-matrix elements on the GPU would double the amount of work offloaded from the CPU. The combination of reduced data transfer and increased work offload should provide a significant performance improvement for OVERFLOW-2.

Table 2. Wall-clock times (ms) measured for CPU cache-based Thomas algorithm (TA) compared to CUDA Cyclic Reduction (CR) using (i) synchronous data transfers, (ii) asynchronous transfers and (ii) asynchronous transfers and kernels for difference grid sizes and data types. (Grid size of 128 with double-precision exceeds the CUDA shared memory limits.) Times are averaged over 200 iterations.

Grid size	Datatype	CPU TA (ms)	$^{(i)}$Sync CR (ms)	$^{(ii)}$Async xfer (ms)	$^{(iii)}$Async kernel (ms)
32	single	32	74		56
32	double	36	115		68
64	single	245	432	543	431
64	double	310	880	864	550
128	single	2020	5460		3610

IV.B. Finite-volume kernel benchmarks

This section presents benchmark results for two kernels commonly used in explicit CFD algorithms. These kernels solve (i) the local and global time-step (δt) (i.e., the maximum stable δt allowed) and (ii) the inviscid flux vector difference. These kernels were implemented within MBFLO3 and compared for accuracy and performance against their equivalent CPU implementations.

MBFLO3[5] is a finite-volume compressible CFD algorithm capable of solving the Navier-Stokes and RANS equations on generalized 3D, multi-block, structured-grids. MBFLO3 employs the Lax-Wendroff time marching algorithm with Ni's distribution formulas. The algorithm is suitable for time-accurate unsteady and steady-state simulations. An implicit dual-time method is available for convergence acceleration. The multi-block algorithm can handle an arbitrary number of block interfaces. Distributed parallel computing is supported with MPI. This CFD algorithm has been used for previous GPGPU acceleration in 2D viscous flow simulations.[6,17]

The benchmarks were conducted on a dual-socket Intel X5677 (quad-core) Xeon server with 12GB DDR3 ECC memory and 4 NVIDIA C2050 GPUs. While the test hardware had multiple GPUs, only single-GPU results are presented.

The first kernel benchmark measures the performance of computing the local and global time-step limit for an inviscid flow over a bump. The geometric configuration and block distribution is shown in Fig. 3. The simplified test configuration allows the grid size to be easily varied for scaling studies.

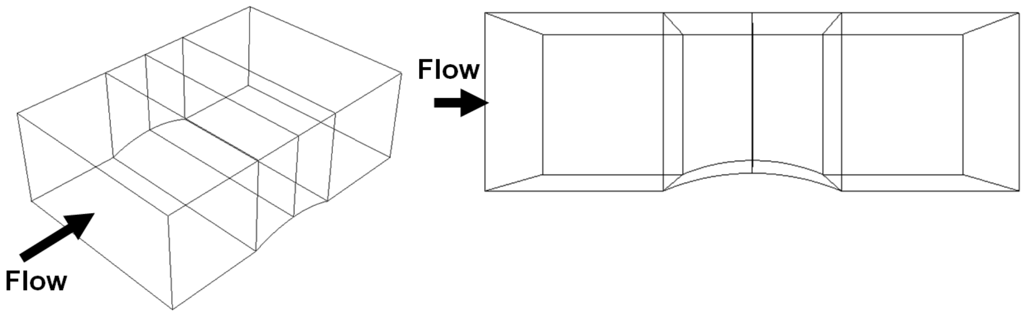


Figure 3. Inviscid bump test configuration for finite-volume kernel benchmarks in MBFLO3.

The local cell-centered time-step calculation within MBFLO3 first requires the interpolation of conserved quantities (i.e., ρ, $\rho\mathbf{u}$, ρe_0) from the grid nodes to the cell centers. This is done by assuming equal weights for all nodal values within each hexagonal element. The velocity ($\mathbf{u}$) and local speed of sound (c) can readily be found from the conservative variables and equation-of-state. For this study, γ was assumed constant allowing a simplified equation-of-state to be used, i.e., $\rho e = p/(\gamma - 1)$. The stable time-step at a cell ijk is approximated using the following formula:

$$\delta t = \mathrm{Vol}/\min(|\mathbf{u}\cdot\mathbf{S}_\xi| + c|\mathbf{S}_\xi|, |\mathbf{u}\cdot\mathbf{S}_\eta| + c|\mathbf{S}_\eta|, |\mathbf{u}\cdot\mathbf{S}_\zeta| + c|\mathbf{S}_\zeta|) \tag{11}$$

where Vol is the cell's volume and $\mathbf{S}_\xi$, $\mathbf{S}_\eta$, $\mathbf{S}_\zeta$ are the face surface area vectors in the ξ, η and ζ grid directions. The surface area vectors in the three coordinate directions are averaged to give a cell-centered value.

The current CUDA implementation is straightforward for this kernel. The 3-d computational grid (or *patch*) is partitioned by cell coordinates into uniform 2-d slices with 32x6x1 cells. These 2-d slices define the thread blocks, i.e., on thread per cell. The thread blocks are mapped onto a larger 2-d grid of blocks for actual kernel execution. Within each kernel, the cell-centered local time-step value is computed in parallel. Once all threads have finished the calculation, a modified binary reduction algorithm, following Harris et al.,[18] is used to compute the thread block's minimum time-step. The reduction kernel is modified to allow thread block sizes that are not powers-of-2. However, the thread block size is limited to the hardware *warp* size to maintain parallel efficiency.[a] Each thread block's minimum δt is then stored in global memory. After all thread blocks have completed, a second minimum reduction kernel is launched to find the patch's global minimum. This single value is returned to the user along with a cell-centered array of local δt values.

Only global memory is used for array access. That is, no shared memory is explicitly used to cache variable access. However, the NVIDIA *Fermi* devices (i.e., CUDA v. 2.0+) provide some automatic data caching that has not been available for previous hardware versions. Future research will investigate the merits of explicit shared memory usage as well as combinations of shared, texture and global memory for optimal performance.

Local time-step performance measurements on multiple grid sizes are shown in Fig. 4. The optimized CPU results are presented alongside the baseline GPU results and GPU results using a *lumped* transfer

[a]Threads within a thread block are partitioned into *warps* in the CUDA SIMT architecture. Each instruction is issued collectively to and executed by the threads in the warp. Currently, a warp is 32 threads.

method. Two different host-to-device transfer methods were implemented: variable-by-variable and lumped-variable approach. The variable-by-variable method issues transfer requests sequentially while the lumped approach transfers groups of variables within a single transfer request. Combining multiple transfers together may reduce latency costs. The results are averaged over 800 iterations. Note, all benchmark results shown used double-precision exclusively.

For this specific benchmark, all variables used by the time-step kernel are transferred from host to device at each iteration including those that are unchanged (e.g., surface area vectors and cell volume). Therefore, the performance presented here can be considered a worst-case scenario.

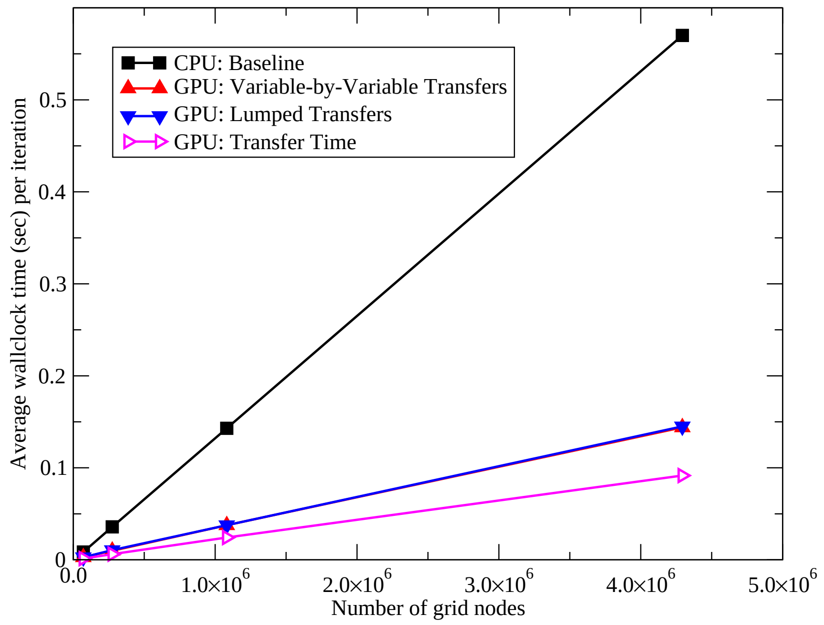


Figure 4. Local stable time-step calculation using finite-volume GPU algorithm compared to CPU baseline for various grid sizes. CPU baseline is shown in black squares; GPU results using variable-by-variable transfers in red triangles; GPU results using *lumped* transfers in blue triangles; and GPU transfers times in purple triangles. Timings are reported in wall-clock seconds and averaged over 800 samples.

The time-step kernel performance results in Fig. 4 show speed-up factors ranging from 3.2–3.9x over the range of grid sizes compared to the baseline CPU tests. The total data transfer time is also shown in Fig. 4. There was only minimal difference observed between the lumped and variable-by-variable transfer approaches so only the lumped transfer times are shown. The transfer time consistently accounted for 63% of the total GPU runtime.

Though the performance improvement realized in the time-step calculation is modest, the results are promising. Reducing the data transfer cost would greatly improve the potential speed-up. For static grids, the cell volumes and face surface area vectors need only be copied to the device once per simulation reducing the transfer overhead by 72%. This could give potential speed-ups approaching 11x on the largest grid. Outright elimination of the transfer overhead could give speed-ups closer to 21x on the largest grid. This scenario could be realized if all calculations were ported to the GPU and any necessary data transfers were overlapped with asynchronous kernel execution via streams. Aside from the data transfer overhead, efficient use of shared and texture memories is expected to further improve the code throughput.

The inviscid fluxes were computed following a strategy similar to the time-step kernel. The face-centered flux vectors were computed by first linearly interpolating the conservative variables from the four surrounding nodal points on each quadrilateral face. The flux differences in each of the three computational directions were executed sequentially with one thread assigned per cell. The flux differences were separated to reduce register pressure and thereby increase the potential occupancy factor.

The inviscid flux benchmark results are shown in Fig. 5. Since no appreciable difference was observed previously, only the lumped GPU transfer approach is shown. Speed-up factors range from 3.5x on the smallest grid up to 4.3x for the two largest grids. The overall computational work required in the flux calculation is approximately twice that of the time-step calculation and this is observed in both the CPU and GPU benchmarks. The higher computational intensity also effectively lowers the data transfer overhead

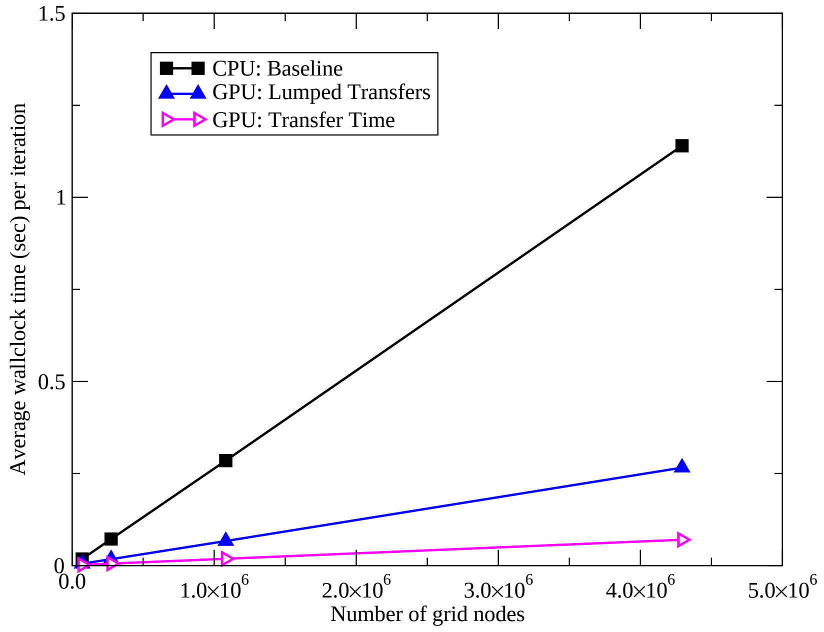


Figure 5. Inviscid flux calculation using finite-volume GPU algorithm compared to CPU baseline for various grid sizes. CPU baseline is shown in black squares; GPU results in red triangles; and GPU transfer times blue triangles. Timings are reported in wall-clock seconds and averaged over 800 samples.

by roughly 2x compared to the time-step kernel. The transfer cost was 32% on the smallest grid and to dropped to 26% on the largest grid size measured. This lower relative cost lessens the potential speed-up that could be realized with reduction or elimination of the memory copies. Total elimination of data transfers on the largest grid could increase the speed-up from 4.3x only to 5.8x. However, the impact of shared and texture memory on the inviscid flux calculation is projected to be greater compared to its impact on the time-step kernel. This is due to the flux kernel requiring 3x as many data loads and interpolations. Repeated loads for this type of compact structured stencil could benefit greatly from explicit caching in shared memory and/or by exploiting automatic texture memory cache functionality.

V. Conclusions

A family of parallel block tridiagonal matrix solvers were implemented in CUDA for GPU acceleration. The block Cyclic Reduction (CR) and Parallel Cyclic Reduction (PCR) algorithms were designed to support a variety of sub-matrix formats and data structures to assess the optimal CUDA implementation.

A performance analysis of the parallel block tridiagonal solvers was conducted using a canonical test problem. The CUDA performance was found to depend significantly upon the sub-matrix element storage method: structure-of-arrays (SOA) versus array-of-structures (AOS). AOS performs optimally on cache-based CPUs while SOA was better suited for vector-based CPUs and CUDA's SIMT architecture. Measurements showed that SOA was 2x faster for CR and 2.6x faster for PCR on CUDA compared to the AOS version when host-device data transfers were ignored. When host-device data transfer was included, even the optimal SOA formulation was nearly 2x slower than the OpenMP implementation of the Thomas Algorithm (TA) due to the overhead cost of transmitting the block matrices.

The parallel CR CUDA solver was then implemented within the OVERFLOW-2 CFD code, validated for correctness and extensively benchmarked using various data transfer methods. The CR CUDA matrix solver was used to invert the block tridiagonal systems generated by the Approximate Factorization Beam-Warming (AF-BW) scheme in the ξ-direction. The left-hand-side (LHS) sub-matrices and right-hand-side (RHS) sub-vectors were all constructed on the CPU. Multiple implementations were tested using synchronous and asynchronous data transfers and kernels with the aid of CUDA *streams*. The time for the CR CUDA method to solve the matrices using double-precision on a 64^3 grid was 2.8x greater than that of the cache-optimized OVERFLOW-2 matrix solver using straightforward synchronized data transfers. An asynchronous method was tested whereby planes of LHS block-matrices and RHS sub-vectors were constructed, transmitted and solved using CUDA streams. This strategy improved the overall performance but was still slower than the CPU version by a factor 1.7x.

These results show that the data transfer overhead will be a limiting factor in solving many CFD-relevant problems on the GPU. This was particularly evident for the block tridiagonal matrix solver within the OVERFLOW-2 code. However, if the transfer cost could be removed, the CUDA-based parallel tridiagonal matrix solvers were faster than the CPU version. In this regard, the next logical stage in the development sequence is to offload the entire AF Beam-Warming algorithm used in OVERFLOW-2 to the GPU. The block tridiagonal matrix solvers presented here would act as an enabling technology by providing an efficient and highly parallel component required by the larger CFD algorithm.

Estimates showed that offloading the entire AF Beam-Warming algorithm to the GPU could be expected to give considerable improvements due to both data transfer reduction and an increase in the floating-point operations done on the GPU. The data transfer was shown to be reduced by over a factor of 17x and the total amount of floating-point operations offloaded from the CPU increased by 2x. Beyond the AF Beam-Warming scheme development, the right-hand-side (RHS) central difference scheme could also be offloaded to the GPU creating a powerful GPU version of the OVERFLOW-2 flow solver.

A second component of this research evaluated CUDA GPU versions of common calculations for compressible flow simulations on structured grids. Kernels for finding the maximum local/global time-step size and computing the inviscid fluxes were implemented in CUDA and tested within the MBFLO3 finite-volume CFD code. Performance improvements, compared to the CPU equivalents, were approximately 4x for both the time-step and flux kernels when including the cost of host-device memory transfers. The host-device data transfer overhead accounted for 72% of the cost in the time-step kernel and approximately 30% for the flux kernel. Reducing or eliminating this overhead, through further GPU kernel porting in conjunction with asynchronous transfers, could lead to speed-ups approaching 21x for the time-step and 6x for the flux kernels.

The finite-volume CFD kernels presented here relied largely upon global memory reads and writes; only the per-block reduction operation exploited the available shared memory hardware. Future research is needed to assess the merits of using shared memory and/or texture memory to increase the throughput. Both the time-step and flux kernels repeatedly accessed common nodal data over the computational cells. The performance could be increased by explicitly caching the surrounding nodal data in shared memory. However, shared memory is a limited resource and must be used judiciously. Texture memory may also be used to increase the performance. GPU texture memory has a separate, dedicated cache that could improve the read-only performance of structured-grid data. The optimal memory management strategy is expected to be a combination of both shared and texture memories. This approach needs to be investigated in future research.

Acknowledgement of Support and Disclaimer

This material is based upon work supported by the United States Air Force under Contract No. FA9550-10-C-0149. Any opinion, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force.

The authors would also like to thank Dr. Fariba Fahroo, AFOSR Program Manager, for her support.

References

[1] www.top500.org.

[2] www.green500.org.

[3] Beam, R. and Warming, R., "An Implicit Finite-Difference Scheme for the Compressible Navier-Stokes Equations," *AIAA Journal*, Vol. 16, 1978, pp. 393–401.

[4] Buning, P., Parks, S., Chan, W., and Renze, K., "Application of the Chimera Overlapped Grid Scheme to Simulation of Space Shuttle Ascent Flows," *Proceedings of the Fourth International Symposium on Computational Fluid Dynamics*, Vol. 1, 1991, pp. 132–137.

[5] Bozinoski, R. and Davis, R., "General Three-Dimensional Navier-Stokes Procedure for Reynolds-Averaged/Detached-Eddy Simulations," *AIAA Paper 2008-756*, 2008.

[6] Phillips, E. H., Zhang, Y., Davis, R. L., and Owens, J. D., "Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units," *AIAA Paper 2009-565, 47th AIAA Aerospace Sciences Meeting and Exhibit*, 2009.

[7]Peaceman, D. and Rachford, H., "The Numerical Solution of Parabolic and Elliptic Differential Equations," *Journal of the Society for Industrial and Applied Mathematics*, Vol. 3, 1955, pp. 28–41.

[8]Lele, S., "Compact Finite Difference schemes with Spectral Like Resolution," *Journal of Computational Physics*, Vol. 103, 1992, pp. 16–42.

[9]Hockney, R. and Jesshope, C., *Parallel Computers 2*, IOP Publishing Ltd., 1988.

[10]Hirshman, S., Perumalla, K., Lynch, V., and Sanchez, R., "BCYCLIC: A Parallel Block Tridiagonal Matrix Cyclic Solver," *Journal of Computational Physics*, Vol. 229, No. 18, 2010.

[11]Zhang, Y., Cohen, J., and Owens, J. D., "Fast Tridiagonal Solvers on the GPU," *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, Jan. 2010, pp. 127–136.

[12]Davidson, A., Zhang, Y., and Owens, J. D., "An Auto-tuned Method for Solving Large Tridiagonal Systems on the GPU," *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, May 2011.

[13]Davidson, A. and Owens, J. D., "Register Packing for Cyclic Reduction: A Case Study," *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, March 2011, pp. 4:1–4:6.

[14]Goddeke, D. and Strzodka, R., "Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 0, No. 0, 2010.

[15]Sakharnykh, N., "Efficient Tridiagonal Solvers for ADI and Fluid Simulation," *GPU Technology Conference*, 2010.

[16]Nichols, R., Tramel, R., and Buning, P., "Solver and Turbulence Model Upgrades to OVERFLOW 2 for Unsteady and High-speed Applications," *AIAA Paper 2006-2824, 24th AIAA Applied Aerodynamics Conference*, 2006.

[17]Phillips, E. H., Davis, R. L., and Owens, J. D., "Unsteady Turbulent Simulations on a Cluster of Graphics Processors," *AIAA Paper 2010-5036, 40th AIAA Fluid Dynamics Conference*, 2010.

[18]Harris, M., Sengupta, S., and Owens, J. D., "Parallel Prefix Sum (Scan) with CUDA," *GPU Gems 3*, edited by H. Nguyen, chap. 39, Addison Wesley, Aug. 2007, pp. 851–876.