 CMES, vol.95, no.5, pp.431-452, 2013

Comparison and Performance Analysis of Multiple CPU/GPU Computing Systems – Resin Infusion Flow Modeling Application

R.H. Haney[1] **and R.V. Mohan**[2]

Abstract: The use of Graphics Processing Units (GPUs) as co-processors for single CPU/GPU computing systems has become pronounced in high performance computing research, however the solution of truly large scale computationally intensive problems require the utilization of multiple computing nodes. Multiple CPU/GPU computing systems bring new complexities to the observed performance of computationally intensive applications, the more salient of which is the cost of local CPU-GPU host and intra-nodal communication. This paper compares and analyzes the performance of a computationally intensive application represented by resin infusion flow during liquid composite molding process for the manufacture of structural composites application via two distinct multiple CPU/GPU computing system architectures. Resin flow infusion modeling during liquid composite molding process is the engineering application of interest in the present study. The global domain is partitioned into a series of sub-domains each of which is solved at the local host and reassembled for the final solution as per the domain decomposition methodology. The candidate application, as with many scientific and engineering applications, uses the Finite Element Method (FEM) to computationally model the governing physics based mass and momentum conversation equations. FEM discretization results in large sparse linear equation systems that are solved iteratively for this class of free surface, moving boundary value problem. Computational analysis software for the GPU environment has been developed using CUDA API for the iterative linear equation system solver based on the preconditioned conjugate gradient method for the solution of linear system of equations. These linear equation systems are solved multiple times with intra-nodal communication utilized, resulting in the converged global system. The interplay of local host CPU/GPU and intra-nodal communication creates mixed performance results for the presented candidate application. The software/hardware factors that affect performance for each architecture are examined and discussed in this paper –

[1] North Carolina A&T State University, Greensboro, NC, U.S.A.
[2] North Carolina A&T State University, Greensboro, NC, U.S.A, For Correspondence.

understanding how the presented candidate application's observed performance is effected by both the individual multiple CPU/GPU computing system architecture and algorithmic/software design is critical to optimize many modern high performance applications which employee the GPU as a hardware accelerator.

Keywords: resin flow infusion modeling, sparse matrix, performance, multiple GPUs.

1 Introduction

The Graphics Processing Unit (GPU) is inexorably becoming common place for the acceleration of computationally intensive applications for single CPU/GPU computing systems [Hamada, Narumi et al. (2009); Kuznik, Obrecht et al. (2010); Corrigan, Camelli et al. (2011); Bustamam, Burrage et al. (2012)] as ever larger computing challenges seek to leverage the dramatic computational power expressed with the modern GPU device [Fatahalian and Houston (2008); Garland, Le Grand et al. (2008); Grozea, Bankovic et al. (2010); Bustamam, Burrage et al. (2012)]. However, truly large scale high performance computing applications necessitate the use of multiple computing nodes. The execution of computationally intensive applications within the context of multiple CPU/GPU computing systems has been shown with varying degrees of success [Karunadasa and Ranasinghe (2009); Corrigan, Camelli et al. (2011); Kim, Seo et al. (2012); Lee and Vetter (2012)]. This paper examines the performance of computational analysis software for resin infusion flow modeling during liquid composite molding process for structural composites via two distinct multiple CPU/GPU computing systems – exposing the effects of different hardware and software influencing factors.

Liquid Composite Molding (LCM) is a popular process used for the manufacture of polymer composite structures. The process involves injection of a polymeric resin inside a mold cavity filled with an oriented, woven dry fiber preform [Mohan, Ngo et al. (1996); Mohan, Shires et al. (2001)]. The resin infusion flow modeling and simulation of the LCM process involves the isothermal process flow solution based on the conservation of resin mass as the governing equation in FEM computational developments. The governing coupled mass and momentum conservation equation is discretized and the fill factors and pressure values are solved in an iterative manner [Mohan, Ngo et al. (1996); Mohan, Shires et al. (2001)]. The computationally intensive portion of the analysis code is from the multiple calls to the linear equation system solver that is solved using the Preconditioned Conjugate Gradient (PCG) method.

The PCG iterative linear equation system solver consists of a set of matrix-vector operations [Shewchuk (1994)] that map well to the natural matrix structure of the

GPU using Nvidia Compute Unified Device Architecture (CUDA) API [Nvidia (2012)]. The performance results of two different multiple CPU/GPU computing systems are examined and discussed, offering possible avenues of further investigation for both rapid and robust code developments using the GPU that can be applied for the candidate application, and translate well to other legacy FEM based codes as well. The present paper is organized as follows.

The GPU and its evolution are briefly discussed in the first section, as well as the details of the CPU and GPU hardware employed in the present work. The next section provides a description of the candidate application and includes the details of the physical problem, model equations, implementation, and the computation solution algorithm involved. A description of the GPU implementation and development for the resin flow infusion modeling is presented in the following section. The final section examines and discusses the computational performance of the presented candidate application providing analysis, comparisons, and conclusions.

2 Graphical Processing Units

The modern GPU consists of heavy concentration of transistors in the Arithmetic Logic Units (ALUs) and wide data bus [Boggan and Pressel (2007); Luebke and Humphreys (2007); Fatahalian and Houston (2008)]. The memory architecture has remained relatively simple to facilitate quicker access to input data. Unlike the CPU [Rumpf and Strzodka (2005); Luebke and Humphreys (2007)], GPU separates the configuration/instruction from the data to be operated on by the processor's large number of ALUs.

2.1 GPU evolution

The gaming industry had been a key driving force for the improvement in the computing power of GPUs over the years [Buck, Foley et al. (2004); Boggan and Pressel (2007); Luebke and Humphreys (2007)]. Images are rendered using matrix operations applied simultaneously to large sets of data at the same time. Graphics hardware receives the instruction/configuration prior to the influx of the data to be operated, a natural Single Instruction Multiple Data (SIMD) environment seen in many parallel systems [Wilkinson and Allen (2005); Fatahalian and Houston (2008)]. Newer and more detailed games required faster processing of matrix operations as well as a wider data bus to enable rapid execution. Transistors were concentrated for floating point operations to facilitate large number of matrix operations [Boggan and Pressel (2007)] inherent in graphical processing hardware. The CPU on the other hand needed to maintain flexibility [Patterson and Hennessy (1998); Silberschatz, Galvin et al. (2003)]. Both CPU and GPU must deal with latency, and do so in distinctly different ways based on their architecture. The CPU

maintains complex memory hierarchy to execute multiple processes in quick succession to deal with latency [Patterson and Hennessy (1998); Silberschatz, Galvin et al. (2003); Boggan and Pressel (2007)]. GPU on the other hand deals with latency by executing large data sets with the same configuration/instruction, which is set up prior to any data processing with a wide data bus and transistors concentrated in the arithmetic region. Fig. 1 illustrates typical CPU and GPU transistor distribution. The evolution of GPU architecture has resulted in a computational workhorse optimized for data throughput.

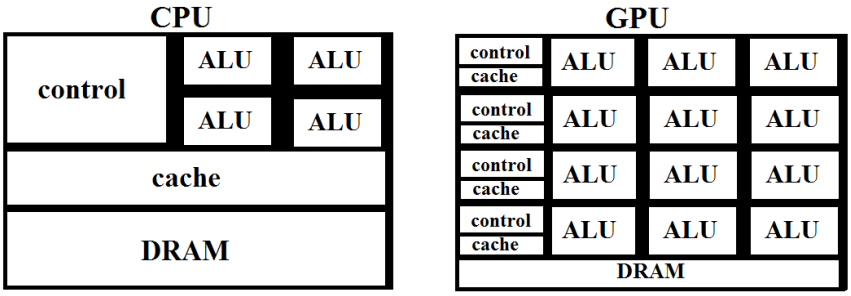


Figure 1: Distribution of transistors for CPU and GPU hardware devices

Initially graphics hardware operated in a fixed pipeline and any programming was low level and very complex [Boggan and Pressel (2007)]. Thus the ability to access the GPU in any realistic way to leverage its computational power was limited. The development of flexible pipeline as well as high level language constructs paved the way for the use of GPU as a co-processor [Boggan and Pressel (2007); Luebke and Humphreys (2007)]. GPU software paradigms are further enabling their use as computational processors in computationally intensive engineering analysis applications such as the one employed in the present work.

2.2 CPU and GPU architectures

In this work, we utilize two distinct multiple CPU/GPU computing systems each consisting of multiple nodes composed of a single commodity GPU and CPU processor for software development and comparative performance analysis. These two multiple CPU/GPU computing systems are denoted as **System A** and **System B** and are defined as follows.

1. ***System A***: The CPU and GPU device architectures for this multiple CPU/GPU computing system can be viewed as the lower-grade of the two presented in this work. Each node in the multiple CPU/GPU computing system contains the following technical specifications.

- *CPU:* A dual-core Opteron processor with a clock frequency of 2.8 GHz, 64-bit DDR2, an L1 cache of 256 KBs, an L2 cache of 2 MBs and a 1000 MHz Front Side Bus.
- *GPU:* Nvidia Quadro FX5600 with 1.5GBs global memory, CUDA compute architecture 1.0, 128 cores, 8192 registers, 16 shared memory banks, a clock frequency of 1.35 GHz and GDDR3 SDRAM using a 384-bit wide bus.

2. ***System B***: The CPU and GPU device architectures for this multiple CPU/GPU computing system can be viewed as the higher-grade of the two presented in this work. Each node in the multiple CPU/GPU computing system contains the following technical specifications.

   - *CPU:* A 6-core Intel X5650 processor with a clock frequency of 2.6 GHz, 64-bit DDR3, an L1 cache of 384 KBs, an L2 cache of 1.5 MBs, an L3 cache of 12 MBs and a 1300 MHz Front Side Bus.
   - *GPU:* Nvidia Tesla M2070 with 6 GBs global memory, CUDA compute architecture 2.0, 448 cores, 32768 registers, 32 shared memory banks, a clock frequency of 1.15 GHz and GDDR5 SDRAM using a 384-bit wide bus.

The next section provides a description of the candidate application and includes the details of the physical problem, model equations, implementation, and the computation solution algorithm involved.

3 Candidate engineering analysis application

Resin infusion flow modeling in liquid composite manufacturing process for the manufacture of woven fabric structural composite parts is the candidate engineering analysis application for the computational code development and GPU, CPU performance analysis employed in the present study. Resin flow infusion modeling involves the tracking of the progressing resin from the injection gates through woven fabric filled Eulerian mold cavity mesh geometry. For thin shell composite structural configurations employing 2D flow representation and thickness averaged 2D flow field, three-node triangular finite elements are used to discretize the mold cavity domain [Mohan, Ngo et al. (1996); Mohan, Shires et al. (2001)]. The finite element discretization for the computational problem involves the Galerkin weighted residual finite element formulation of the resin mass conservation equation in conjunction with the Darcian porous media flow field representing the momentum conservation, and an implicit methodology for the transient solution [Mohan, Ngo et al. (1996), Mohan, Shires et al. (2001)].

3.1 Resin infusion flow modeling

Following the discussions in [Mohan, Shires et al. (2001); Mohan, Ngo et al. (1996)], the resin flow through the fiber preform contained within the mold cavity is represented by a transient mass conservation equation. The physical mass conservation equation (formed by coupling the mass conservation equation with the momentum equation via Darcian velocity field) is given by Eq. (1) with $\bar{K}$ the permeability tensor, μ the resin viscosity, P the pressure field, and Ψ the state variable, representing the resin flow infusion state of the mold cavity region.

$$\frac{\partial}{\partial t}\int_{\Omega}\Psi\partial\Omega \equiv \int_{\Omega}\nabla\left(\frac{\bar{K}}{\mu}\nabla P\right)\partial\Omega \tag{1}$$

Further details of this above governing equation are available in [Mohan, Ngo et al. (1996)] and [Mohan, Shires et al. (2001)]. The value of the state variable $\Psi$is 0 in the regions of the mold where the resin has not infused the fiber preform and 1 where the resin has completely infused the fiber preform in any given region of the Eulerian mold cavity domain Ω in the FEM computations.

As discussed in [Mohan, Ngo et al. (1996)] and [Mohan, Shires et al. (2001)], the application of the Galerkin weighted residual formulation and approximating for the pressure $P$and fill factor $\Psi$with appropriate elements to define the mold cavity resin infusion geometry, and associated shape functions yields a semi-discrete system of equations given by Eq. (2) with $C$the lumped mass matrix, $K$the stiffness matrix, $q$the load vector, and $\dot{\Psi}$the time derivative.

$$C\dot{\Psi} + KP = q \tag{2}$$

The transient semi-discrete equation is then solved by introducing the finite difference approximation given by Eq. (3) for the time derivative term.

$$\dot{\Psi} = \frac{\Psi^{n+1} - \Psi^{n}}{\Delta t} \tag{3}$$

3.2 Resin flow infusion modeling strategy

The semi-discrete Eq. (2) can be reduced to Eq. (4), as discussed in [Mohan, Ngo et al. (1996)] and [Mohan, Shires et al. (2001)].

$$C_{ii}\Psi_i^{n+1} - C_{ii}\Psi_i^{n} + \Delta t K_{ij}P_j = \Delta t q_i \tag{4}$$

The above form of the discretized equation is solved during the resin infusion flow modeling during the transient resin progression at each time-step. Equation (4)

defines the implicit form of the process flow modeling in LCM, as shown in references [Mohan, Ngo et al. (1996)] and [Mohan, Shires et al. (2001)]. The generalized algorithm for the finite element computations for the process flow modeling and analysis of the resin progression in the transient, free surface flow problem for each time step is summarized in **Algorithm 1**. The finite element based solution strategy employed results in a system of linear equations that is solved using an iterative pre-conditioned conjugate gradient method involving matrix-vector, and dot product operations.

Algorithm 1: Implicit Pure FE methodology for Resin Infusion Flow Modeling Computation

(For progression from time step n to time step $n+1$and iteration m)
1. **REPEAT**
2. **SET** $\{\Psi_i\}_m^{n+1}$to$\{\Psi_i\}^n$ (save previous fill factor values)
3. **CALL** assembleC for C_i (assembleC forms lump mass matrix)
4. **CALL** assembleK for K_{ij} (assembleK forms stiffness matrix$K$)
5. **CALL** assembleLoad on q_i (assembleLoad forms load vector$q$)
6. **REPEAT**
7. **SET** boundary conditions on K_{ij} (Modified load vector$g$)
8. **SET** $\{g_i\}_m$ to $C_{ii}\{\Psi_i\}^n - C_{ii}\{\Psi_i\}_m^{n+1} + \Delta t\{q_i\}_m$
(Where $\hat{K}_{ij}$is Kmatrix with boundary conditions applied)
9. **SOLVE** $\left\lfloor \hat{K}_{ij} \right\rfloor_m \left\{P_j\right\}_m = \{g_i\}_m$
(Compute new nodal resin fraction field using equation (4))
10. **SET** $C_{ii}\{\Psi_i\}_{m+1}^{n+1} = C_{ii}\{\Psi_i\}^n - \Delta t\,[K_{ij}]\left\{P_j\right\}_m + \Delta t\{q_i\}_m$
11. **IF** $\left\| C_{ii}\{\Psi_i\}_{m+1}^{n+1} - C_{ii}\{\Psi_i\}_m^{n+1} \right\| \leq \xi$ **THEN**
12. **BREAK**
13. **ELSE**
14. **SET** $\{\Psi_i\}_m^{n+1}$to $\{\Psi_i\}_{m+1}^{n+1}$
15. **ENDIF**
16. **UNTIL** mass resin convergence
17. **UNTIL** all nodes are filled

The interested reader is referred to [Mohan, Ngo et al. (1996)] for further details on the resin infusion flow modeling in LCM process and the finite element discretization employed for the computational problem.

3.3 CPU and GPU resin flow infusion modeling validation

The validity of the CPU and GPU code developments for multiple computing nodes was determined by careful examination of the numerical results against the analytical solution for a simple 2D circular plate radial injection model (see Fig. 2). The analytical solution for this simple circular plate geometry radial injection condition for comparison were determined based on the radial flow front location, and injection port pressure at time step t for multiple numbers of sub-domains derived by partitioning the initial global domain and applying equations presented in reference [Mohan, Ngo et al. (1996)]. Eq. (5) and (6) define the radial flow front and injection port pressure respectively with R_0 the inner radius, μ the resin viscosity, ϕ the fiber volume fraction, $\overline{K}$ the permeability, and H as the element or mold cavity thickness.

$$R(t) = \sqrt{\frac{Qt}{\pi\phi H} + R_0^2} \tag{5}$$

$$P_0 = \left[\frac{\mu Q}{2\pi \bar{K} H} \ln\left(\frac{R(t)}{R_0}\right)\right] \tag{6}$$

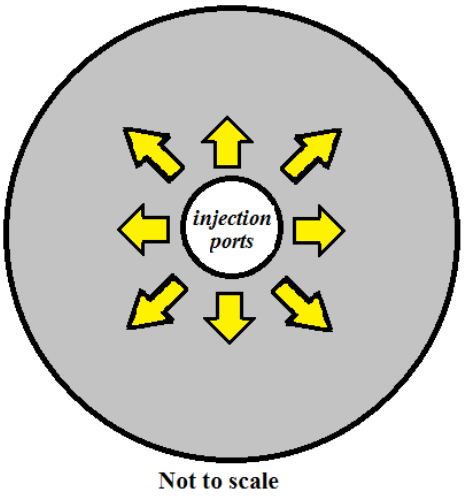
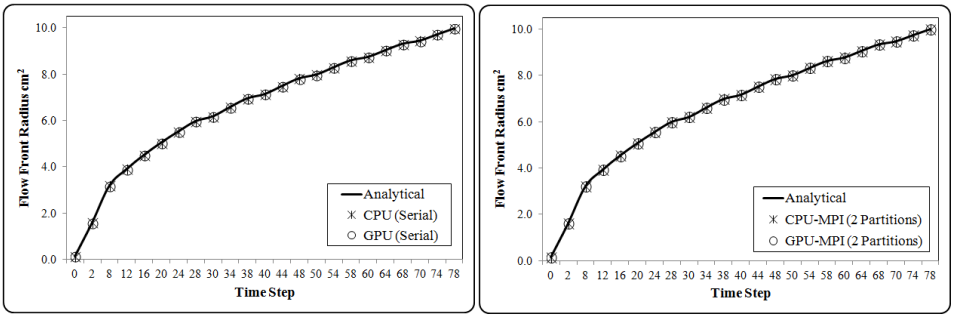


Figure 2: 2D circular plate radial injection flow model

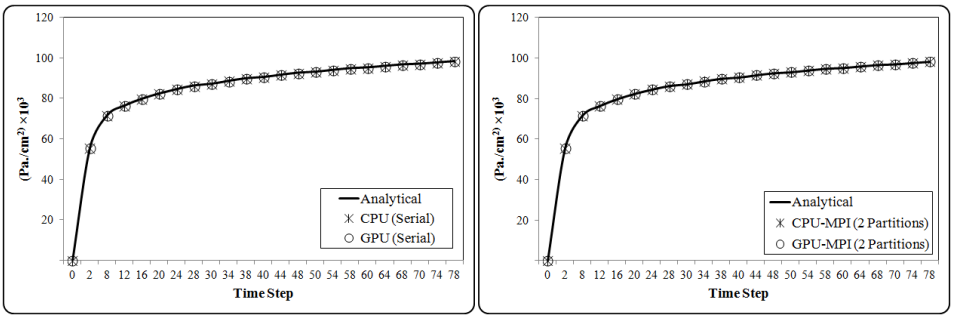
The radial flow front and injection port pressures for this thin shell circular plate model from the computational CPU and GPU analysis values for serial and 2-

partitions were compared to the analytical values. Fig. 3, Fig. 4, Fig. 5 and Fig. 6 clearly demonstrate the congruency of CPU and GPU numerical solutions to the analytical solutions at each time step for serial and multiple partitions for both **System A** and **System B**. Additional computational modeling analysis comparisons for other complex composite structures also showed excellent numerical congruency between CPU and GPU analysis code execution runs.



(a) Analytical to serial CPU/GPU **(b) Analytical to multiple partition CPU/GPU**

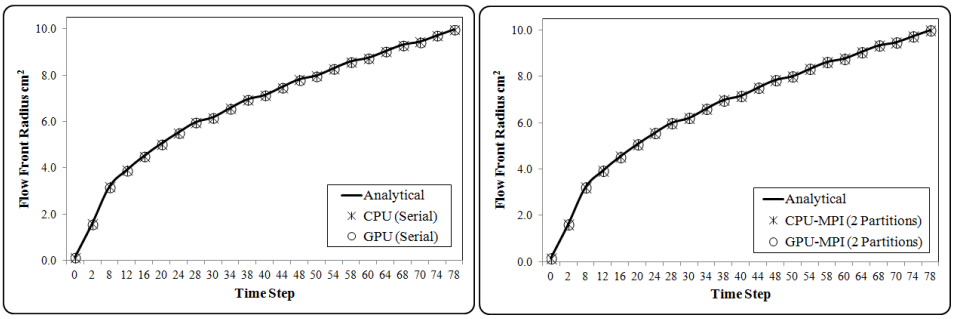
Figure 3: Transient flow front locations comparisons for serial/multiple CPU/GPU System A



(a) Analytical to serial CPU/GPU **(b) Analytical to multiple partition CPU/GPU**

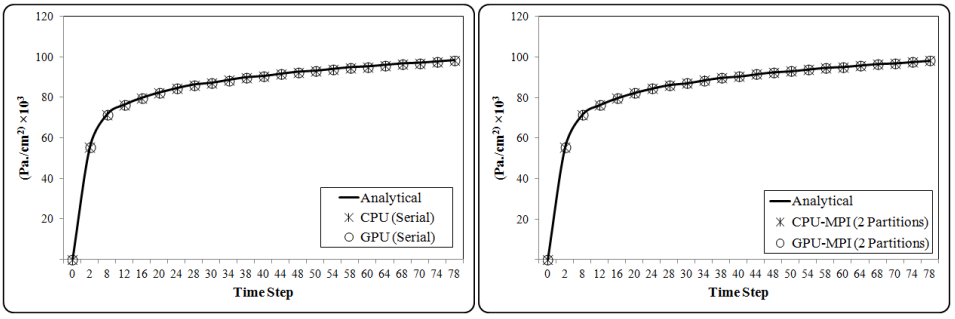
Figure 4: Transient injection pressure comparisons for serial/multiple CPU/GPU System A

The mapping of the candidate application to the multiple GPU/CPU computing system environments is discussed next.



(a) Analytical to serial CPU/GPU (b) Analytical to multiple partition CPU/GPU

Figure 5: Transient flow front location comparisons for serial/multiple CPU/GPU System B



(a) Analytical to serial CPU/GPU (b) Analytical to multiple partition CPU/GPU

Figure 6: Injection pressure comparisons for serial/multiple CPU/GPU System B

4 GPU implementation and code development strategies

Finite element discretization of the transient, moving boundary value problem involves the solution of linear system of equations multiple times during the computational analysis and is the computationally intensive portion in the resin flow infusion modeling analysis of complex composite structures. The Preconditioned Conjugate Gradient (PCG) solution methodology for the solution of linear system of equations (line 9 of **Algorithm 1**) executes matrix operations, involves significant computational load, and was chosen as the key resin flow infusion modeling kernel to port to the GPU. The PCG solver is composed of sparse matrix-vector multiplication and vector operations which is shown in **Algorithm 2**.

The PCG solver is executed with an element-centric [Mohan, Ngo et al. (1996); Mohan, Shires et al. (2001)] rather than node-centric focus which results in a larger

Algorithm 2: Preconditioned conjugate gradient (solves $Ax = b$)
Input: Matrix A and load/force vector b
Output: solution vector x

1. **Set** $r_0 \Leftarrow b - Ax_0$
2. **Set** $z_0 \Leftarrow M^{-1} r_0$
3. **Set** $p_0 \Leftarrow z_0$
4. **Set** $k \Leftarrow 0$
5. **DO UNTIL CONVERGENCE**
6. $\alpha_k \Leftarrow \frac{r_k^T z_k}{p_k^T A p_k}$
7. $x_{k+1} \Leftarrow x_k + \alpha_k p_k$
8. $r_{k+1} \Leftarrow r_k - \alpha_k A p_k$
9. **IF** $(\|r_k - r_{k+1}\| \leq \xi)$ **BREAK**
10. $z_{k+1} \Leftarrow M^{-1} r_{k+1}$
11. $\beta_k \Leftarrow \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}$
12. $p_{k+1} \Leftarrow z_{k+1} + \beta_k p_k$
13. $k \Leftarrow k + 1$
14. **END DO**

ratio of computation to communication cost [Haney (2006)] – an optimal strategy to leverage the dramatic floating point power of the GPU. The sparse matrix-vector operation (line 2 and 10 of **Algorithm 2**) is the largest computational cost of the PCG solver [Buatois, Caumon et al. (2009); Helfenstein and Koko (2011)] and an obvious section for porting to the GPU device. The other major computational portions of the linear system solver are vector updates and scalar dot-products, both of which are handled via the CUBLAS library calls [Shewchuk (1994); Nvidia (2007)].

The presented candidate application for this study utilize domain decomposition to partition a global problem domain such that each resulting sub-domain is solved with a local CPU/GPU host – communication is handled via the MPI. The interplay of the coarse-grained parallelism expressed by MPI and the fine-grained parallelism of the GPU is the source of often disappointing performance of computationally intensive applications employing multiple CPU/GPU computing systems [Papadrakakis, Stavroulakis et al. (2011); Wang, Huang et al. (2011); Song and Dongarra (2012)].

5 Computational performance analysis and comparison

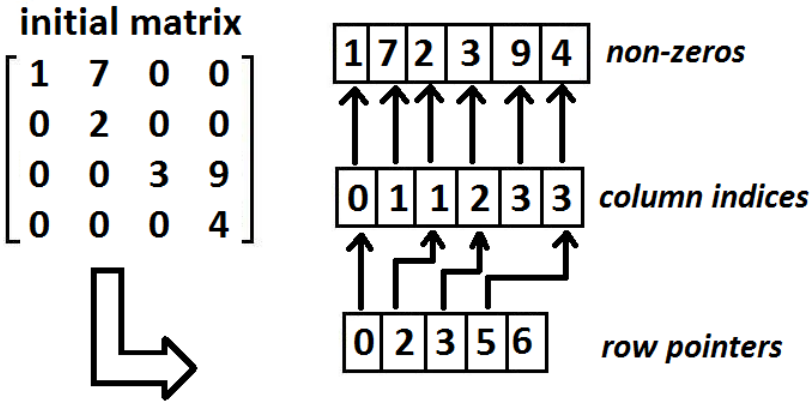


Figure 7: Compressed Sparse Row (CSR) format

The first half of this section produces the preliminary data for the models being examined and the second half of this section presents the observed performance using **System A** and **System B** multiple CPU/GPU computing systems. The input unstructured mesh models are initially solved using the CSR (Compressed Sparse Row) compression format shown in Fig. 7 then the BCSR2x2 (Block Compressed Sparse Row) compression format is utilized. The CSR compression format is well documented as a potential performance problem as increased pointer indirection can magnify irregularity of data element access for sparse matrix-vector multiplications [Baskaran and Bordawekar (2008); Hugues and Petiton (2010)]. The BCSR2x2 compression format, shown in Fig. 8, has been shown to increase locality in cases where dense sub-blocks exist in the sparse matrices [Baskaran and Bordawekar (2008); Hugues and Petiton (2010)], such as with linear equations generated for the presented candidate application – effectively isolating the influence of spatial locality for both **System A** and **System B** architectures.

5.1 Preliminaries

The computational performance of the GPU and CPU code developments and their comparative performance were studied employing two separate unstructured finite element mesh models representing resin infusion in a composite structural part configuration. For the computational performance comparisons, this resulted in two computational models composed of differing numbers of nodes with three-node triangular elements but with the same geometries (Fig. 9). The unstructured

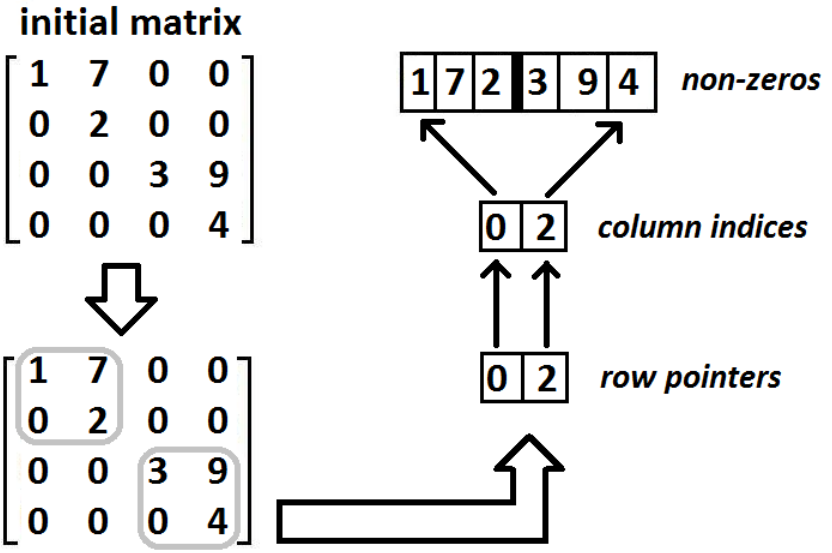


Figure 8: Block Compressed Sparse Row with 2x2 sub-blocks (BCSR2x2) format

mesh models are labeled as follows.

- Model **MA**: Model consisting of 26,936 nodes and 53,148 triangular elements.
- Model **MB**: Model consisting of 103,196 nodes and 204,970 triangular elements.

Computational performance benchmarking for the GPU and CPU developments and resin flow infusion modeling was accomplished as follows.

- **Speedup factor**: The ratio of CPU execution time to GPU execution time whereby the larger the value, the more optimal the performance obtained through GPU.

5.2 Observed performance and analysis – System A

Executing with the CSR format we observe that MPI provides a distinct performance boost over the serial versions for this multiple CPU/GPU computing system regardless of the input unstructured meshes. However, when using MPI in combination with the local GPU we get less distinct benefit over MPI without employing

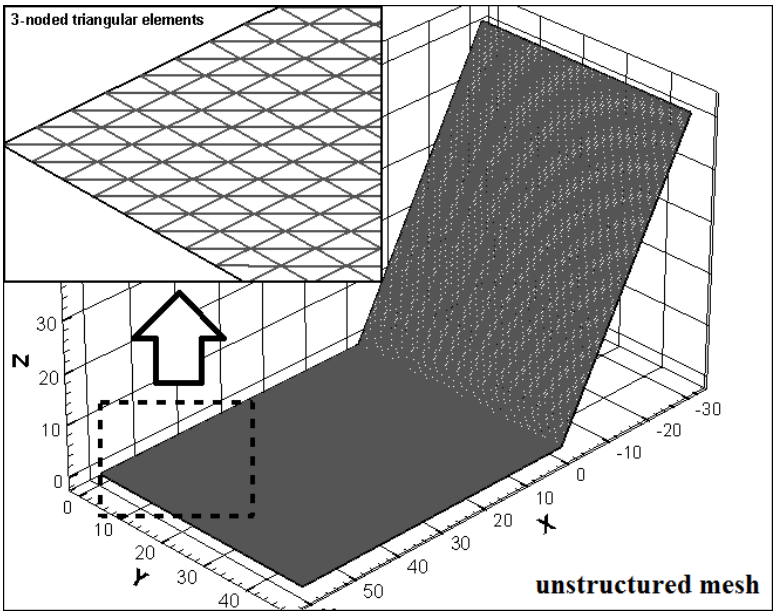


Figure 9: Unstructured mesh geometry that represents both MA and MB models

the local GPU. Fig. 10 shows the smaller unstructured mesh model **MA** and Fig. 11 shows the larger unstructured mesh model **MB**. Table. 1 and Table. 2 are the observed full solution times for the **MA** and **MB** models respectively.

Table 1: System A execution time for mesh MA

Partitions	MPI + GPU Time (secs.)	MPI + CPU Time (secs.)	Speedup Factor
2	1,604.200	1,666.960	1.034
4	1,357.890	825.989	0.608
16	1,461.590	260.669	0.178

The unstructured mesh model **MA** exposes a nearly constant total solution time when using MPI and the local GPU regardless of the number of partitions, whereas MPI with no GPU displays a nearly linear decrease in this time as shown in Fig. 10. The unstructured mesh model **MB** displays a slight performance boost when using MPI and the local GPU for partition counts less than 16 as the larger mesh will inexorably utilize more of the GPU computational resources to mitigate latency – higher probability of memory address coalescing with the CUDA-enabled hardware.

Table 2: System A execution time for mesh MB

Partitions	MPI + GPU Time (secs.)	MPI + CPU Time (secs.)	Speedup Factor
2	16,985.600	20,169.400	1.187
4	8,570.300	9,633.960	1.124
16	23,956.100	2,495.580	0.104

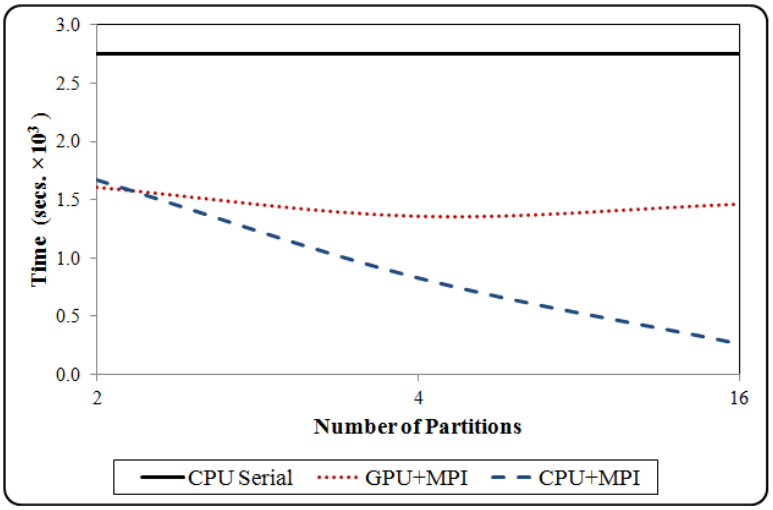


Figure 10: Total solution time for mesh MA with CSR compression using System A

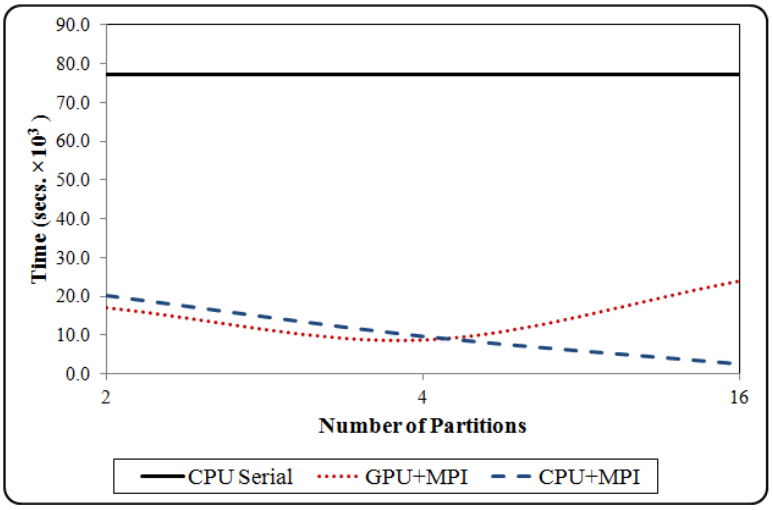


Figure 11: Total solution time for mesh MB with CSR compression using System A

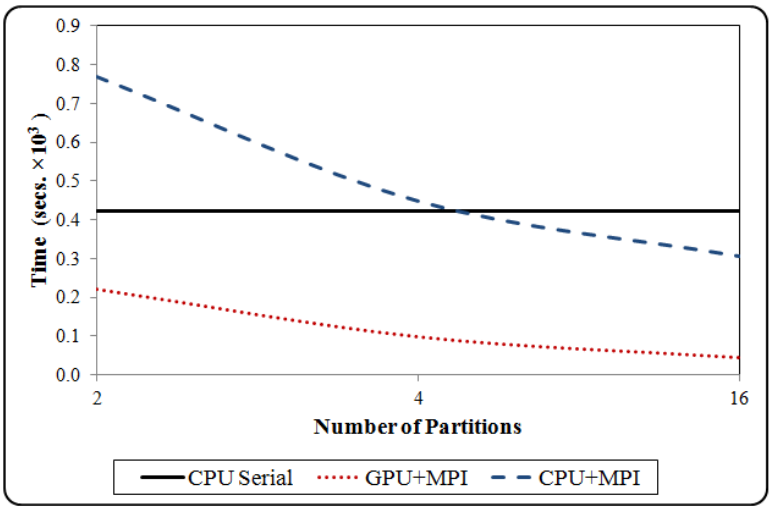


Figure 12: Total solution time for mesh MA with CSR compression using System B

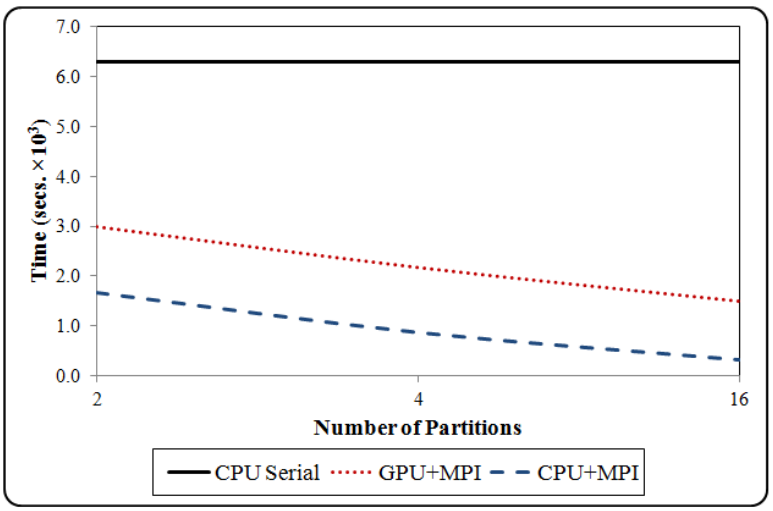


Figure 13: Total solution time for mesh MB with CSR compression using System B

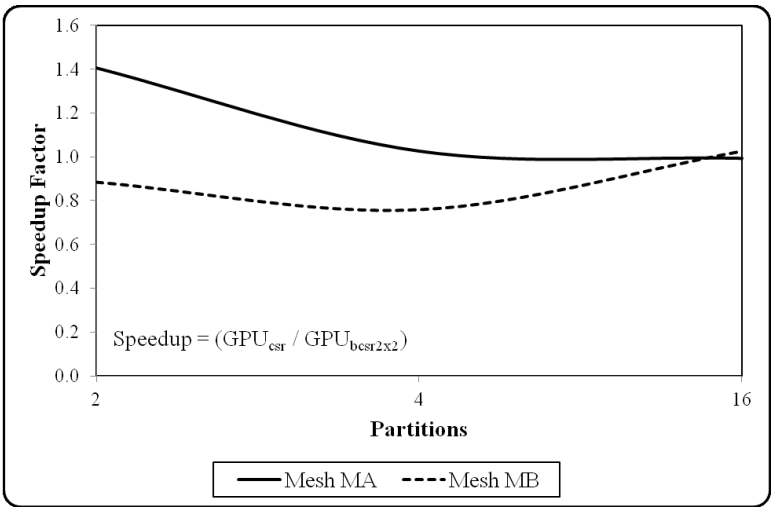


Figure 14: Mesh MA and MB with CSR and BCSR2x2 compression - System A

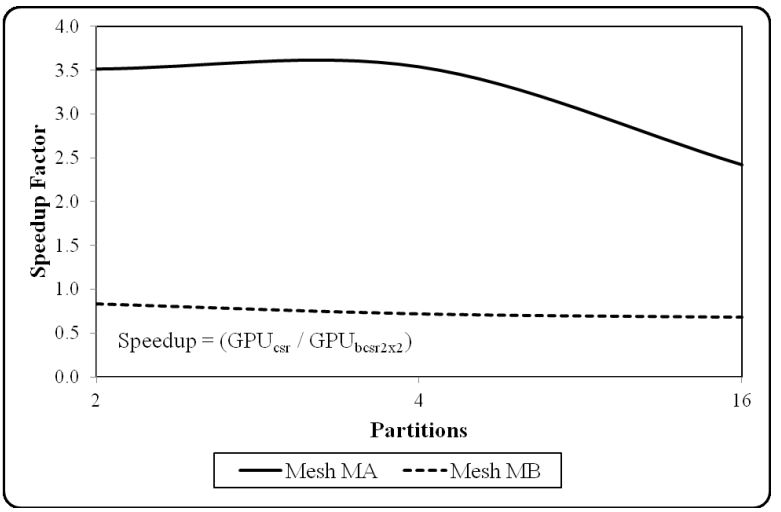


Figure 15: Mesh MA and MB with CSR and BCSR2x2 compression - System B

The dramatic decrease in performance for MPI employing local host GPU after 4 partitions correlates well with the inefficiency of the fine grained parallelism of the GPU to cooperate with the coarse grained parallelism defined by domain decomposition methodology as implemented by the MPI standard [Papadrakakis, Stavroulakis et al. (2011); Wang, Potluri et al. (2011)] – i.e., the GPU cannot directly access the communication buffers of the MPI calls and must cross the local CPU-GPU communication bus as defined by PCIe and this is a well documented bottleneck in GPGPU computing [Luebke (2008); Lee, Kim et al. (2010); Rehman (2010)].

Executing with the BCSR2x2 compression format, as shown in Fig. 14, spatial locality appears to be a larger factor in performance for the smaller model **MA** with partitions less than 16 when intra-nodal communication assumes a larger cost of execution. The larger model **MB** has a different behavior with the smaller partitions yielding a lower performance and the higher partition showing better performance – as seen in Fig. 14.

5.3 Observed performance and analysis – System B

Executing with the CSR compression format we observe the performance of the input unstructured mesh models for this multiple CPU/GPU computing system to be the relative inverse of that seen with **System A**. The larger unstructured mesh model **MB** performs worse for the MPI and local GPU paradigm (Fig. 13) and the smaller mesh model **MA** performs better using MPI and the local GPU (Fig. 12). This behavior is counter-intuitive as previous experience with GPU enhanced applications have shown increased performance benefit for problems requiring larger floating-point operations [Boggan and Pressel (2007); Baskaran and Bordawekar (2008); Grozea, Bankovic et al. (2010); Bustamam, Burrage et al. (2012)]. Table. 3 and Table. 4 are the observed full solution times for the **MA** and **MB** models respectively.

Table 3: System B execution time for mesh MA

Partitions	MPI + GPU Time (secs.)	MPI + CPU Time (secs.)	Speedup Factor
2	220.462	767.364	3.481
4	98.034	448.665	4.577
16	44.392	307.081	6.917

Given that the input mesh models have the same parameters and geometries for **System B** and **System A**, the inverted performance of the candidate application solved with the defined input meshes using **System B** are likely due to a faster memory

Table 4: System B execution time for mesh MB

Partitions	MPI + GPU Time (secs.)	MPI + CPU Time (secs.)	Speedup Factor
2	2,995.870	1,675.850	0.559
4	2,172.520	872.019	0.401
16	1,490.310	318.895	0.214

I/O for the local GPU. Both **System A** and **System B** have 384-bit wide bus, but **System B** uses a GDDR5 memory I/O device whereas **System A** employees the GDDR3 memory I/O device. The GDDR5 can execute on both edges of the clock pulse and thus push the input data to the processing cores of **System B** GPU at a higher rate than **System A**, exposing the significant latency generated as a product of the local CPU-GPU host and intra-nodal communications.

Executing the candidate application using the BCSR2x2 compression format, as shown in Fig. 15, spatial locality appears to provide a nearly consistent advantage with the smaller input unstructured mesh model **MA** – regardless of number of partitions. The larger mesh model **MB** illustrates consistent performance degradation – immune to any number of partitions used.

6 Concluding Remarks

The multiple CPU/GPU computing systems examined in this work illustrate the sometimes deleterious effects that iterative sparse matrix solvers can present when MPI and the local GPU device are combined, illustrating the critical importance of understanding the role of both hardware and software in high performance applications. In the general case, MPI that employees either CPU or GPU as the local backend device will produce a performance benefit for either **System A** or **System B** – as shown in Fig. 10, Fig. 11 and Fig. 12, Fig. 13 respectively. However the advantage of using MPI with the local GPU over MPI without the local GPU is not as discernible as this depends heavily on the specific hardware and algorithmic approaches employed.

This work has illustrated that later generations of GPU devices, such as **System B**, have mitigated much of the spatial locality issues that plagued earlier devices executing sparse matrix-vector operations with necessary data compression but has revealed other concerns as shown by comparing Fig. 12 and Fig. 13.. The Faster memory I/O employed by **System B** increases the potential process throughput, defined as the central paradigm of latency mitigation for GPU devices, but can actually create paradoxical situations where increased computational loads can degrade application performance. Software/algorithmic factors can mitigate poor applica-

tion performance to a degree, as with increasing spatial locality via a compression format that load near-by elements of the matrix to on-board GPU device registers as is the case with the 2x2 blocks of BCSR2x2 compression.

Critical to the optimal performance of computationally intensive applications for multiple CPU/GPU computing systems is the understanding of the underlying architecture and the application to execute.

Acknowledgement: The support in part by contracts/grants from Office of Naval Research and Clarkson Aerospace is acknowledged.

References

Baskaran, M. M.; Bordawekar, R. (2008): Optimizing Sparse Matrix-Vector Multiplication on GPUs, IBM Research: 11.

Boggan, S. K.; Pressel, D. M. (2007): GPUs: An Emerging Platform for General-Purpose Computation. Aberdeen Proving Ground, MD, USA, Army Research Lab: 50.

Buatois, L.; Caumon, G.; Levy, B. (2009): Concurrent number cruncher: a GPU implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems*, vol. 24, no. 3, pp. 18.

Buck, I.; Foley, T.; Horn, D.; Sugerman, J.; Fatahalian, K.; Houston, M.; Hanrahan, P. (2004): Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH,* vol. 23, no. 3, pp. 777-786.

Bustamam, A.; Burrage, K.; Hamilton, N. A. (2012): Fast Parallel Markov Clustering in Bioinformatics Using Massively Parallel Computing on GPU with CUDA and ELLPACK-R Sparse Format. *IEEE/ACM Trans Comput Biol Bioinform.*, vol. 3, no. 9, pp. 13.

Corrigan, A.; Camelli, F. F.; Lohner, R.; Wallin, J. (2011): Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids*, vol. 66, no. 2, pp. 221-229.

Fatahalian, K.; Houston, M. (2008): GPUs: A Closer Look. *ACM Queue,* vol. 6, no. 2, pp. 10.

Garland, M.; Le Grand, S.; Nickolls, J.; Anderson, J.; Hardwick, J.; Morton, S.; Phillips, E.; Zhang Y.; Volkov, V. (2008): Parallel Computing Experiences with CUDA . *Micro, IEEE,* vol. 28, no. 4, pp. 13-27.

Grozea, C.; Bankovic, Z.; Laskov, P. (2010): FPGA vs. Multi-Core CPUs vs. GPUs: Hands-on Experience with a Sorting Application. *Facing the multicore-*

challenge Berlin, Heidelberg, Springer-Verlag Berlin, Heidelberg pp. 105-117.

Hamada, T.; Narumi, T.; Yasuoka, K.; Nitadori, K.; Taiji, M. (2009): *42 TFlops Hierarchical N-body Simulations on GPUs with Applications in both Astrophysics and Turbulence*. The International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM New York, NY, USA.

Haney, R. H. (2006): *Study and Evaluation of Domain Decomposition Approaches in two Parallel Software Code Developments for Process Flow Modeling in Liquid Composite Molding*. Master of Science, North Carolina A & T State University.

Helfenstein, R.; Koko, J. (2011): Parallel preconditioned conjugate gradient algorithm on GPU. *Journal of Computational and Applied Mathematics,* vol. 236, no. 15, pp. 6.

Hugues, M. R.; Petiton, S. G. (2010): *Sparse Matrix Formats Evaluation and Optimization on a GPU*, IEEE Computer Society Washington, DC, USA.

Karunadasa, N. P.; Ranasinghe, D. N. (2009): Accelerating High Performance Applications with CUDA and MPI. *2009 International Conference on Industrial and Information Systems (ICIIS)*. Sri Lanka, IEEE, pp. 331-336.

Kim, J.; Seo, S.; Lee, J.; Nah, J.; Jo, G.; Lee, J. (2012). *SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters*. International Conference on Supercomputing, ACM New York, NY, USA.

Kuznik, F.; Obrecht, C.; Rusaouen, G.; Roux, J.-J. (2010): LBM based flow simulation using GPU computing processor. *Computers & Mathematics with Applications,* vol. 59, no. 7, pp. 12.

Lee, S.; Vetter, J. S. (2012): *Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing*. The International Conference for High Performance Computing, Networking, Storage, and Analysis, IEEE Computer Society Press Los Alamitos, CA, USA.

Lee, V. W.; Kim, C.; Chhugani, J.; Deisher, M.; Kim, D.; Nguyen, A. D.; Satish, N.; Smelyanskiy, M.; Chennupaty, S.; Hammarlund, P.; Singhal, R.; Dubey, P. (2010): *Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU*.

Luebke, D. (2008): *CUDA: Scalable parallel programming for high-performance scientific computing*. Biomedical Imaging: From Nano to Macro, 2008, Paris, ACM New York, NY, USA.

Luebke, D.; Humphreys, G. (2007): How GPUs Work. *IEEE Computer, 2007*.

Mohan, D. R.; Shires, D.; Mark, A. (2001): Scalable Large Scale Process Modeling and Simulations in Liquid Composite Molding. *Computational Science - ICCS 2001*, Springer Berlin Heidelberg, pp. 1199-1208.

Mohan, R. V.; Ngo, N. D.; Tamma, K. K.; Fickie, K. D. (1996): On a Pure Finite-Element-Based Methodology for Resin Transfer Mold Filling Simulations, Army Research Lab, pp. 18.

Nvidia (2007): CUDA CUBLAS Library: Version 1.1, Nvidia Corporation, pp. 84.

Nvidia (2012): NVIDIA CUDA C Programming Guide: Version 4.2, Nvidia Corporation, pp. 173.

Papadrakakis, M.; Stavroulakis, G.; Karatarakis, A. (2011): A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures. *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 13-16, pp. 1490-1508.

Patterson, D. A.; Hennessy, J. L. (1998): *Computer Organization & Design: The Hardware/Software Interface*. San Francisco, California, USA, Morgan Kaufmann Publishers, Inc.

Rehman, M. S. (2010): *Exploring Irregular Memory Access Applications on the GPU*. Master of Science, International Institute of Information Technology.

Rumpf, M.; Strzodka, R. (2005): Graphics Processor Units: New Prospects for Parallel Computing. *Numerical Solution of Partial Differential Equations on Parallel Computers. A. M. a. T. Bruaset, Aslak, Springer.* vol. 51, pp. 89-134.

Shewchuk, J. R. (1994): An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Pittsburgh, PA, USA, Carnegie Mellon University, pp. 64.

Silberschatz, A.; Galvin, P.; Gagne, G. (2003): *Applied Operating System Concepts: windows XP update*, John Wiley & Sons, Inc.

Song, F.; Dongarra, J. (2012): *A Scalable Framework for Heterogeneous GPU-Based Clusters*. ACM Symposium on Parallel Algorithms and Architectures, ACM New York, NY, USA.

Wang, H.; Potluri, S.; Luo, M.; Singh, A. K.; Sur, S.; Panda, D. K. (2011): MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science - Research and Development,* vol. 26, no. 3-4, pp. 257-266.

Wang, L.; Huang, M.; Narayana, V.; El-Ghazawi, T. A. (2011): *Scaling Scientific Applications on Clusters of Hybrid Multicore/GPU Nodes*. Computing Frontiers Conference, ACM New York, NY, USA.

Wilkinson, B.; Allen, M. (2005): *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Upper Saddle River, NJ, USA, Pearson Education, Inc.