Implementing the Himeno Benchmark with CUDA on GPU Clusters

Everett H. Phillips and Massimiliano Fatica
NVIDIA Corporation
Santa Clara, California, United States
ephillips@nvidia.com, mfatica@nvidia.com

***Abstract*—This paper describes the use of CUDA to accelerate the Himeno benchmark on clusters with GPUs. The implementation is designed to optimize memory bandwidth utilization. Our approach achieves over 83% of the theoretical peak bandwidth on a NVIDIA Tesla C1060 GPU and performs at over 50 GFlops. A multi-GPU implementation that utilizes MPI alongside CUDA streams to overlap GPU execution with data transfers allows linear scaling and performs at over 800 GFlops on a cluster with 16 GPUs. The paper presents the optimizations required to achieve this level of performance.**

I. Introduction

The Himeno benchmark [1] was developed by Dr. Ryutaro Himeno in 1996 at the RIKEN Institute in Japan. Since its introduction the benchmark has grown in popularity and is used throughout the HPC community, especially in Japan. Linpack, the well known benchmark used for ranking the fastest 500 computer in the world, is highly compute intensive, often approaching a large percentage of the theoretical floating point throughput on large computer systems. The Himeno benchmark by contrast is highly memory intensive, bound by memory bandwidth on modern processors.

A. The Himeno benchmark

The Himeno benchmark focuses on the solution of a 3D Poisson equation in generalized coordinates on a structured curvilinear mesh:

$$\frac{\partial^2 p}{\partial x^2}+\frac{\partial^2 p}{\partial y^2}+\frac{\partial^2 p}{\partial z^2}+\alpha\frac{\partial^2 p}{\partial xy}+\beta\frac{\partial^2 p}{\partial xz}+\gamma\frac{\partial^2 p}{\partial yz}=\rho \quad (1)$$

Several numerical methods (like the fractional step method) used to solve the incompressible Navier-Stokes equations require the solution of such equation. With the processing time dominated by the Poisson solution, it makes the Poisson procedure a good measure of overall performance.

Using finite differences, the Poisson equation is discretized in space yielding a 19-point stencil as shown in figure 1.

The discretized Poisson equation is solved iteratively using Jacobi relaxation. Table I shows the main solver loop which applies the stencil to the pressure array P. The 10 arrays $a0-3$, $b0-2$, and $c0-2$ store the spatially varying weights used in the stencil that account for the local grid geometry. The $wrk2$ array is a temporary storage for the updated pressure values, bnd is a boundary condition flag used to specify whether a grid location is solid or fluid, and $wrk1$ is the source term that appears on the right hand side of the Poisson equation.

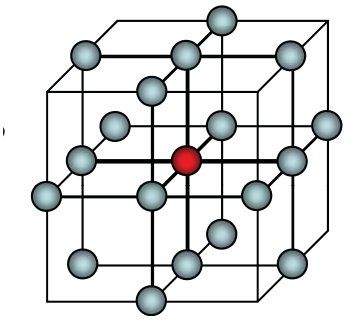

Figure 1. Stencil for the discretization of the Poisson equation in generalized coordinates.

The benchmark is designed to run on a wide range of computer systems. Table II lists the sizes of the benchmark along with the memory footprint of each case.

Problem Size		Memory size
XS	32 x 32 x 64	3.5 MB
S	64 x 64 x 128	28 MB
M	128 x 128 x 256	224 MB
L	256 x 256 x 512	1792 MB
XL	512 x 512 x 1024	14336 MB

Table II
Himeno benchmark standard domain sizes and memory footprints.

II. GPU architecture and CUDA

The GPU architecture has now evolved into a highly parallel multi-threaded processor with very high floating point performance and memory bandwidth. NVIDIA's

978-1-4244-6443-2/10/$26.00 ©2010 IEEE

```
for (i=1; i<imax-1; i++)
   for (j=1; j<jmax-1; j++)
      for (k=1; k<kmax-1; k++)
{
 s0 =  a0[i][j][k]* p[i+1][j][k]
     + a1[i][j][k]* p[i][j+1][k]
     + a2[i][j][k]* p[i][j][k+1]
     + b0[i][j][k]*(p[i+1][j+1][k] - p[i+1][j-1][k]
                  - p[i-1][j+1][k] + p[i-1][j-1][k])
     + b1[i][j][k]*(p[i][j+1][k+1] - p[i][j+1][k-1]
                  - p[i][j-1][k+1] + p[i][j-1][k-1])
     + b2[i][j][k]*(p[i+1][j][k+1] - p[i+1][j][k-1]
                  - p[i-1][j][k+1] + p[i-1][j][k-1])
     + c0[i][j][k]* p[i-1][j][k]
     + c1[i][j][k]* p[i][j-1][k]
     + c2[i][j][k]* p[i][j][k-1]
     + wrk1[i][j][k];
 ss = (s0 * a3[i][j][k]-p[i][j][k])
     * bnd[i][j][k];                    //(ss = delta P)
 wrk2[i][j][k]=p[i][j][k]+omega*ss;     //(over-relaxation)
 gosa += ss*ss;                         //(residual, measure of convergence)
}
```

Table I
COMPUTATIONAL KERNEL: THE OUTPUT IS THE PRESSURE (WRK2) AT THE NEW ITERATION AND THE RESIDUAL TO MONITOR CONVERGENCE.

Tesla 10-series, a product line for high performance computing, has GPUs with 240 cores and 4 GB of memory. The PCI-e card (C1060) has a clock frequency of 1.296 GHz and a 1U system with 4 cards (S1070) has a clock frequency of 1.44 GHz. The cards have a 512–bit GDDR3 memory interface with a 800 MHz memory clock frequency that gives $512/8(bytes) * 2(DDR) * 800\,Mhz = 102\,GB/s$ theoretical peak memory bandwidth.

The GPU is especially well-suited to address problems that can be expressed as data-parallel computations, i.e. the same program is executed on many data elements in parallel. CUDA is a parallel programming model and software environment designed to expose the parallel capabilities of GPUs. CUDA extends C or Fortran by allowing the programmer to define functions or subroutines, called kernels, that when called are executed on the GPU by potentially thousands of parallel threads.

III. CUDA IMPLEMENTATION

Before starting the discussion on the CUDA aspects of the porting, let us analyze the benchmark, in particular a quantity called Compute Intensity (CI), the ratio between memory accesses and floating point operations. Referring to the code listing in Table I, at each spatial location there are 14 floating point values that must be accessed, and a total of 34 floating point operations. This results in a CI of 34 flops / 14*(4 bytes) = 0.607 flops/byte. The algorithm is bandwidth limited and since only one of the 14 arrays is reused, its also very cache-unfriendly.

As a result, it is clear that the CUDA porting strategy should focus on maximizing the effective memory bandwidth. The solution strategy must expose enough parallelism to completely utilize the GPU's resources while reducing the amount of redundant data accesses.

The key to reducing redundant data access is using the on-chip shared memory to store the pressure data. Shared memory is a user-managed cache that allows threads within the same thread-block to cooperate with one another. At 16 KB per multiprocessor, the shared memory is a scarce resource and must be used sparingly. A larger block of data will improve efficiency of the algorithm by having a lower surface-area-to-volume-ratio. However if a single block uses all of the shared memory available, no other blocks can be scheduled to run concurrently on the same multiprocessor, which lowers the hardware's ability to hide memory latency. Thus, there is an inevitable trade-off in the 3D data partitioning scheme.

In previous work [4], the 3D domain is decomposed into 3D sub-blocks processed by a 2D block of threads. The 2D thread block loads and processes several planes of the input data, writing results directly to the global memory.

In our approach we dramatically reduce the shared memory usage by foregoing the storage of the entire 3D data block. Instead, similar to [7], we store only three planes which contain all the neighbor data necessary

to process a plane of the sub-block. These three planes serve as a cyclic buffer, which are swapped as we march along in the Z-direction, loading a new plane and then computing the output for the current plane.

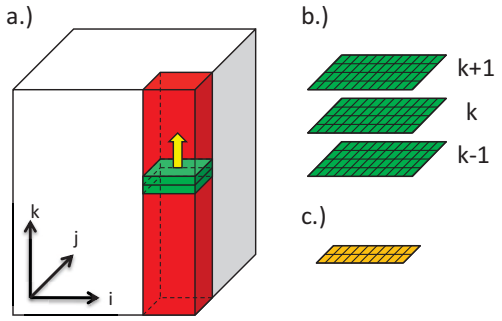

Figure 2. Thread blocks process columns of the domain using a cyclic buffer of 3 planes in shared memory: a) data access of a thread block, b) the three shared-memory planes, c) the thread-block

Since we only store 3 planes we are able to run several thread blocks concurrently on each multiprocessor and have virtually no limit to the Z-direction size of the data block. In fact, as we increase the size of the data block in the Z-direction the algorithm becomes more efficient since fewer redundant planes are loaded. We take this approach to the extreme, strictly decomposing the domain in 2D and processing an entire column of planes with a single thread block, as shown in fig. 2.

Another advantage to the marching cyclic buffer approach is the computation of the residual, which is used to monitor the convergence of the iterative solver. Since each thread will compute an entire column there is no need to store the result of the residual at each spatial location. Instead, each thread accumulates a partial sum as it processes a column of the domain. One could then write these partial sums to the global memory, which are only a single plane in size, and then sum these with an additional pass over these values with a parallel reduction kernel.

We further reduce the memory bandwidth and storage required for the residual by adding a parallel reduction in shared memory to the end of our Jacobi iteration kernel. We reuse one of the 3 shared memory planes to sum the partial column residuals into a single value. This results in writing only one value for each thread block, instead of one value for each thread. After this optimization the residual computation and summation is virtually free.

In the following sections we will present several optimization steps and their performance implications:

- Coalescing the memory access
- Optimizing the execution configuration

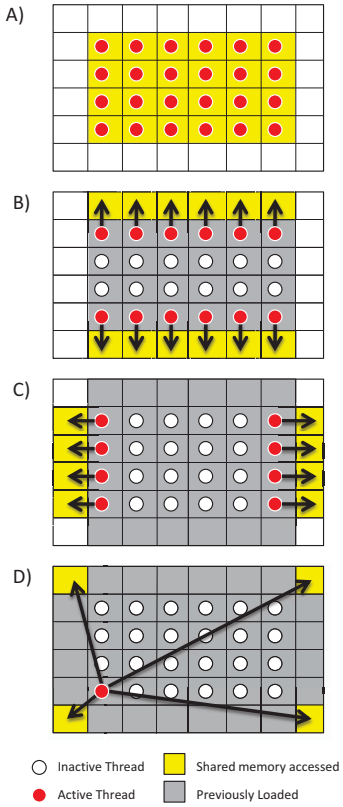

Figure 3. A block of threads loads a plane of the pressure array into shared memory in several steps. A) interior points, B) top and bottom halo points, C) left and right halo points, D) corner points.

- Using the texture cache
- Removing logic and branching

The initial implementation was performing at 22.4 GFlops.

A. *Coalescing memory transactions*

The GPU memory subsystem has some special requirements to achieve optimal bandwidth (see [2] for a detailed description of these rules). Similar to CPU cache lines, GPU memory is partitioned into segments that can be loaded in a single memory transaction. On current hardware the memory controller can fetch 128, 64, and 32 byte segments. For optimal performance each warp of 32 threads should access data in a single segment. This means we must pad the size of array rows to a multiple of 32, and pad the beginning of each array to ensure the 32 values accessed are aligned with the segments.

Coalescing the memory access is very effective because it affects all 14 of the 3D arrays. This simple optimization improves performance by over 57%, from 22.4 to 35.2 GFlops.

```
__global__ void jacobi_kernel( ... ){
  ...
__shared__ float p_sh[3][(BLOCK_Y+2)][(BLOCK_X+2)];
//Load first 2 Planes
p_sh[btm]...
p_sh[mid]...
for(int z=1; z<ZMAX; z++) {
     index += plane_size;
     //Load next pressure plane
     p_sh[top] = ...
     ...
     //apply stencil
     s0  = a0*p_sh[top][ty+1][tx+1] + a1*p_sh[mid][ty+2][tx+1] ...
     ss = bound*(s0*a3 - p_sh[mid][ty+1][tx+1]);
     //write result
     if( inside the domain )  wrk2[index] = p_sh[mid][ty+1][tx+1] + omega*ss;
  //swap planes  top --> mid --> btm --> top
  }
  }
```

Table III
CUDA CODE FOR JACOBI KERNEL

B. Execution Configuration

Another important and easy optimization is determining the best thread-block dimensions. Large blocks have more efficient data access since the halo or ghost cell region will be a smaller percentage; however, this also means each block will use more shared-memory and registers.

The CUDA toolkit comes with an occupancy calculator to assist in determining optimal block size. Occupancy is a measure of GPU utilization as a ratio of the number of a kernel's threads capable of concurrently executing on a multiprocessor relative to the maximum possible.

The toolkit also comes with a visual profiler that provides an abundant amount of useful information for tuning the performance of GPU programs, including occupancy, instruction throughput, memory bandwidth, and many other performance counters.

In experiments, best performance is achieved with 64x3 blocks, matching the predictions obtained with simple bandwidth efficiency and occupancy calculations, shown in table IV .

With this additional optimization, the performance is now 42.3 GFlops.

C. Texture Cache

Further improvements are made by loading P with texture fetches. Texture memory provides cached read-only access that is optimized for spatial locality.
The load statements in the kernel:

```
p_sh[btm][ty  ][tx  ] = p[index];
```

are simply replaced with tex1Dfetch statements as shown below:

```
p_sh[btm][ty  ][tx  ] =
          tex1Dfetch(ptex, index);
```

Using the texture cache prevents redundant loads of global memory. When several blocks request the same region the data will be loaded from the cache. It is important to notice that texture cache is not designed to reduce latency, thus texture loads have similar cost to global loads regardless of whether or not there is a cache hit.

The Texture implementation reaches 47.1 GFlops.

D. Removing Logic

As stated above, texture cache does not reduce latency, however, the use of texture allows for more flexible data access patterns and consolidates redundant acceesses. Thus, our final optimization is to exploit this to reduce logic expressions and the total number of instructions to load the Pressure array. The load of the P array is typically done in several steps, as shown in figure 3. First, all points excluding halo regions are loaded. Next, threads on the left and right boundaries of the block load the left and right halo points, then threads on the top and bottom edges load the top and bottom halo. Finally, a single thread loads the four corner points.

The corresponding code is shown in table V.

However, after employing the texture cache, the access pattern can be altered to remove all if statements from the pressure loads. This results in loading the

block x	block y	Shared memory(Bytes)	Registers	Occupancy	Load Efficiency	Blocks per Multi-processor
16	16	3888	6400	0.500	0.296	2
32	8	4080	6400	0.500	0.400	2
64	4	4725	6400	0.500	0.444	2
64	3	3960	4800	0.563	0.400	3
64	2	3168	3200	0.500	0.333	4

Table IV
COMPARING SEVERAL EXECUTION CONFIGURATIONS (THREAD-BLOCK SIZES) TO FIND A BALANCE OF OCCUPANCY, LOAD EFFICIENCY, AND CONCURRENT BLOCKS PER MULTIPROCESSOR.

```
//load next plane
//interior points
p_sh[top][ty+1][tx+1] = p[index];
//top and bottom halo
if(ty==0      )  p_sh[top][0 ][tx+1] = p[index-rsize];
if(ty==BLOCK_Y-1)p_sh[top][(BLOCK_Y+1)][tx+1] = p[index+rsize];
//left and right halo
if(tx==0      )  p_sh[top][ty+1][0 ] = p[index-1];
if(tx==BLOCK_X-1)p_sh[top][ty+1][(BLOCK_X+1)] = p[index+1];
//corener points
if(tx==0&&ty==0)
 {
    p_sh[top][0 ][0 ] = p[index-rsize-1];
    p_sh[top][0 ][(BLOCK_X+1)] = p[index-rsize+BLOCK_X];
    p_sh[top][(BLOCK_Y+1)][0 ] = p[index+BLOCK_Y*rsize-1];
    p_sh[top][(BLOCK_Y+1)][(BLOCK_X+1)] = p[index+BLOCK_Y*rsize+BLOCK_X];
  }
__syncthreads();
```

Table V
LOADING P, WITH LOGIC

pressure plane in only 4 instructions as shown in table VI.

This change not only improves performance, but also makes the code much cleaner and easier to read.

The code is now performing at 51.2 GFlops.

E. Effect of the optimization steps

Figure 4 shows the effect of the progressive optimization steps. We went from the first naive implementation running at 22.4 GFlops to the fully optimized version running at 51.2 Gflops. Using the Compute Intensity factor of 0.607, this is equivalent to 84.3 GB/s of bandwidth, 83% of the available peak.

F. Previous work

The implementations used in two previous papers on the Himeno benchmark on GPUs ([4] and [5]) are different from the one described in this paper.

In the work performed at Titech ([4]), the problem was decomposed in blocks of shape $(16 \times 16 \times 8)$. Inside each block, 256 threads (arranged as 16×16) read 8 elements each. All the elements of the pressure array were loaded in shared memory: this required 12.5 KB of shared memory and translated in a single block scheduled per multiprocessor. The measured memory bandwidth was 67.9 GB/s on a GeForce GTX280 (out of the theoretical peak of 142 GB/s). They also implemented a multi-GPU version using Tesla S1070. On the XL-sized benchmark, they achieved 524 GFlops on 16 GPUs and 709 GFlops on 32 GPUs.

Naruse ([5], [6]) used a block of shape $(64 \times 4 \times 64)$. Inside each block, 256 threads (arranged as 64×4) read 64 elements each. The pressure was loaded only for the 3 planes necessary to compute the stencil (similar to our approach) resulting in a shared memory usage of only 4.7 KB and in the possibility of scheduling 3 blocks per multiprocessor. While the single GPU implementation has performance comparable to our implementation in terms of percentage of peak memory bandwidth , the multi-GPU implementation did not scale as well as ours: the performance went from 70 GFlops with a GTX 285 GPU to 170 Gflops with 4 GPUs.

```
//load next plane
p_sh[top][ty  ][tx  ] = tex1Dfetch(ptex, index+psize-rsize-1);
p_sh[top][ty  ][tx+2] = tex1Dfetch(ptex, index+psize-rsize+1);
p_sh[top][ty+2][tx  ] = tex1Dfetch(ptex, index+psize+rsize-1);
p_sh[top][ty+2][tx+2] = tex1Dfetch(ptex, index+psize+rsize+1);
__syncthreads();
```

Table VI
LOADING FROM TEXTURE CACHE

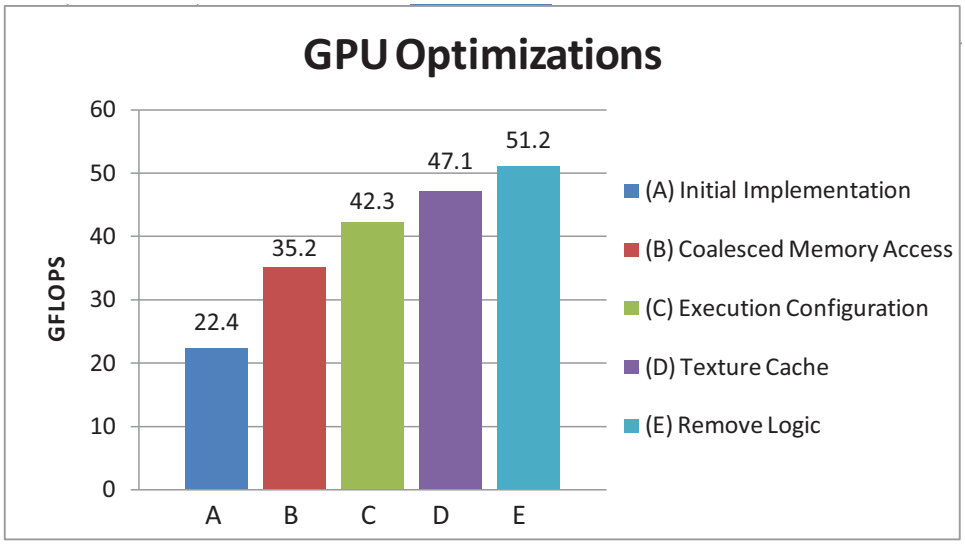

Figure 4. GPU performance after each optimization. The completely optimized version runs at up to 51.2 GFLOPS on a Tesla C1060 GPU.

G. Performance of the GPU implementation

A workstation with a single C1060 GPU reaches 51.2 GFlops or 84.3 GB/s of sustained performance. As seen in figure 5 our implementation outperforms the latest generation of quad-core CPUs (Nehalem) by a factor of $10\times$ and previous generation CPUs (Harpertown) by over $33\times$.

H. Accuracy of the GPU implementation

The CPU implementation of the Himeno benchmark uses single precision. The benchmark could be run on different classes of data sets, from the small (S) class on a $32 \times 32 \times 64$ grid to the extra large (XL) class on a $512 \times 512 \times 1024$ grid.

During the validation phase, we noticed a discrepancy between the CPU solution and the GPU solution when the grid size was increased. The original CPU code is using a single accumulator.

The main source of error is due to the growing difference in magnitude between the running sum and the elements added to it. In this particular case, all the elements of the sum are positive, so there are no cancellation errors.

Accurately computing very long sum of floating point values is a very well known problem in numerical analysis ([3]): to increase the accuracy of the final result, one could resort to Kahan summation, use multiple accumulators (for example with a tree reduction) or use double precision representation for the running sum(s).

The CUDA implementation computes the residual using a tree reduction. This method is more accurate since partial sums are typically much closer in magnitude: the small elements are summed into larger elements which are summed together, avoiding the truncation errors associated with summing values of disparate magnitudes.

To minimize the changes to the CPU code, we used double precision with the single accumulator to get a reference solution. As we can see in Table VII, the GPU results are very close to the reference implementation.

IV. MULTI-GPU IMPLEMENTATION

Extending the solver to multiple GPUs requires further decomposing the domain into portions for each GPU. The simplest approach is to slice the domain along the Z-direction. This 1-D decomposition keeps all the boundary data for communication in contiguous

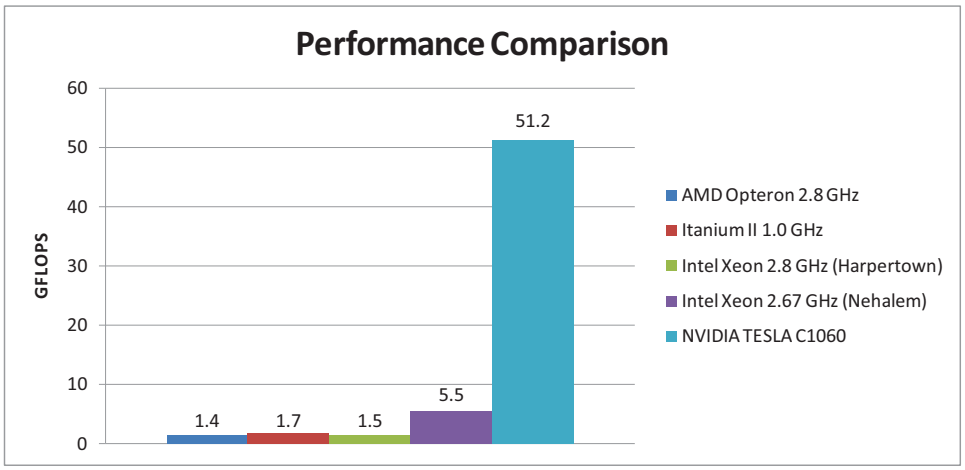

Figure 5. Performance comparison between a single C1060 GPU and several traditional processors

I	J	K	CPU double precision	CPU single precision	CPU % error	GPU single precision	GPU % error
32	32	64	6.714e-3	6.711e-3	0.039	6.713e-3	0.005
64	64	128	3.417e-3	3.408e-3	0.285	3.416e-3	0.049
128	128	256	1.723e-3	1.768e-3	2.608	1.722e-3	0.095
256	256	512	8.683e-4	4.883e-4	43.77	8.637e-4	0.533

Table VII
CONVERGENCE OF THE SOLVER AFTER A SINGLE ITERATION: COMPARISON WITH DOUBLE PRECISION ACCUMULATION ON CPU, SINGLE PRECISION ACCUMULATION ON THE CPU AND SINGLE PRECISION PARALLEL ACCUMULATION ON THE GPU

regions of memory. Each domain must exchange the top and bottom planes with top and bottom neighbors, and use the data to fill a halo area above and below the local domain, as shown in figure 6.

In a naive multi-GPU implementation, each GPU would compute the solution for the subdomain, then boundary data would be transferred to the Host, which would use MPI to exchange data with neighbors, and finally copy received data back to the GPU. This is less than ideal since the GPU will be idle during communication. A better approach is to factor the computations such that they can be carried out concurrently with communications. This improves the performance and allows for improved scalability on GPU clusters.

There are two types of communications taking place, GPU to CPU over the PCI-Express bus, and CPU to CPU communications using MPI. Overlapping the PCI-e communications with computations can be accomplished in CUDA using streams with appropriate memory allocations and transfer calls.

The multi-GPU code also has room for improvements. The initial implementation does not take into consideration that there are 2 GPUs on each physical node. Thus, some communications will be performed using the network, while others simply pass through system memory. In our test system the two computation halves, A and B, are arranged such that A is communicated over the network, while B communication goes through system memory. It would be ideal to split A and B into unequal portions so the slower communication can be overlapped with a larger amount of computation, while the faster communications can be executed concurrently with a smaller portion.

A few extra kernels were created to allow for an uneven split between A and B, and are shown to be effective in fine tuning the parallel performance. In general, since A communication takes place over the network, SPLIT should be chosen such that $B > A$.

A. GPU Cluster Performance

We performed benchmarks on an 8-node cluster with 16 GPUs. Each node is connected to half of a Tesla S1070 system, containing 4 GPUs, so that each node is connected to 2 GPUs. Each node has 2 Intel Xeon E5462 (2.8 GHz with 1600 MHz FSB) and 16 GB of memory. The nodes are connected with SDR Infiniband.

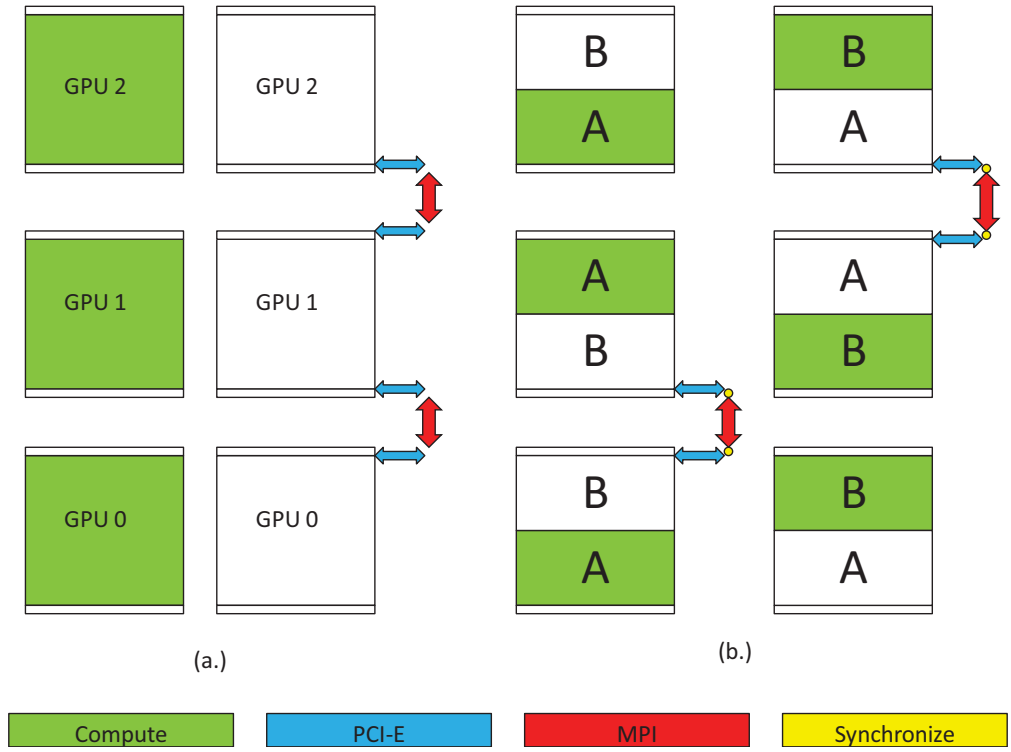

Figure 6. Multi-GPU decomposition strategies. (a.) naive non-overlapping communication, (b.) overlapping communication strategy. In (b.) the A section is computed while B is communicated, then B is computed, while A is communicated

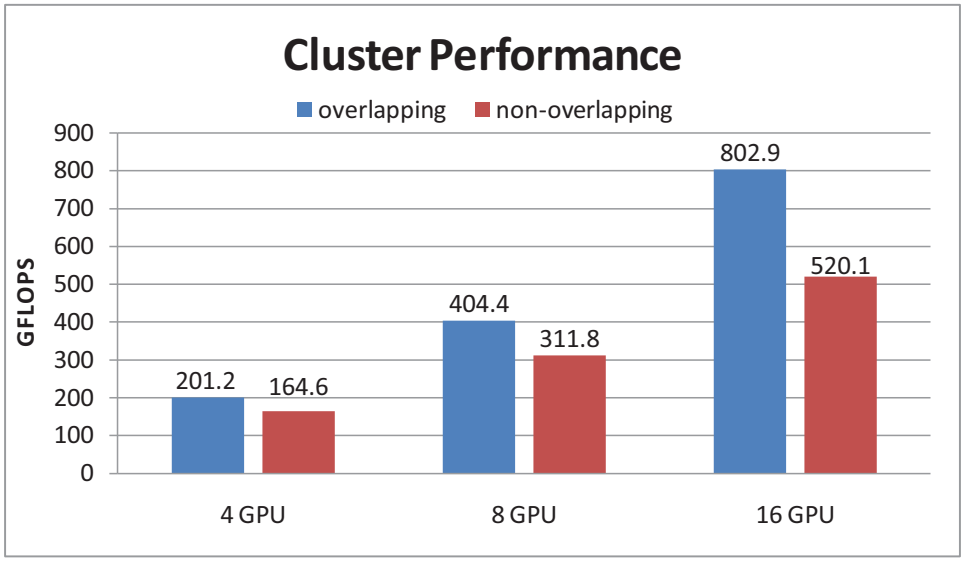

Figure 7. Performance of Himeno benchmark before and after communications overlapping on various GPU cluster configurations for the XL size class.

Figure 7 shows the cluster performance on the XL problem size for 4, 8, and 16 GPU configurations with and without the communication overlapping strategy. After hiding the data transfers and communication cost the cluster scales linearly reaching over 800 GFlops or 1.3 TB/s sustained performance. The small 8U GPU cluster outperforms all but one of the top performing systems listed on the Himeno website [1], as seen in

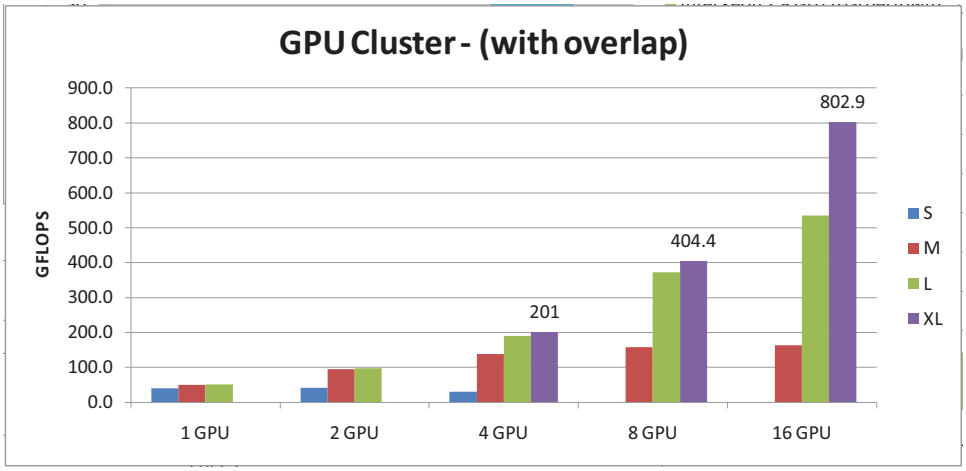

Figure 8. Performance of Himeno benchmark of varying size on various GPU cluster configurations with computation/communication overlap optimization

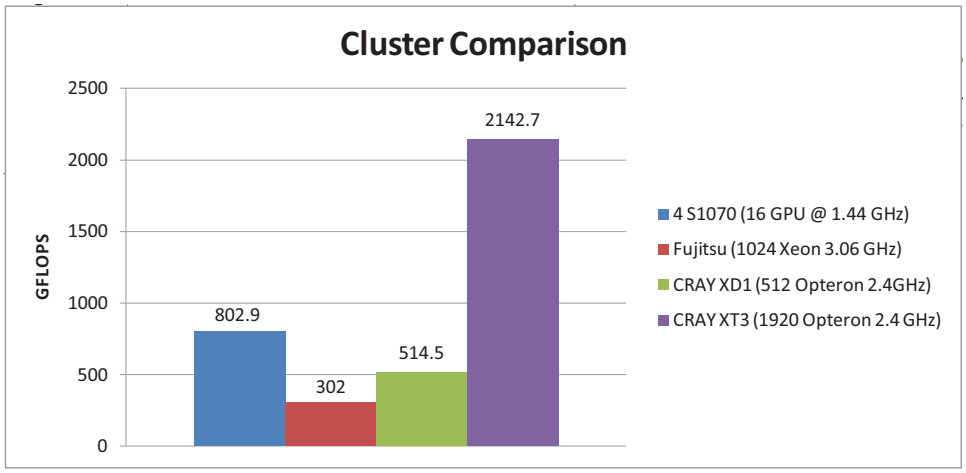

Figure 9. CPU and GPU cluster comparisons (data obtained from Himeno website [1])

figure 9. Compared to the CRAY XT3 which scores over 2 TFlops with 1920 Opteron CPUs, each GPU in our cluster matches the performance of 44 Opteron CPUs.

V. Conclusions

With CUDA and Tesla GPUs, we boosted the Himeno performance of both a workstation and cluster. GPUs, with their floating point performances and excellent memory bandwidth, are very well suited to tackle computational problems. The use of GPUs allows one to reduce the number of host nodes required to reach a target performance level and can significantly reduce the cost of high performance interconnects.

References

[1] The Riken Himeno CFD Benchmark: http://accc.riken.jp/HPC/HimenoBMT/index_e.html

[2] NVIDIA CUDA Compute Unified Device Architecture Programming Guide

[3] N. Higham, "The Accuracy of Floating Point Summation", SIAM J. Sci. Comput., 14(4):783-799, July 1993.

[4] S. Matsuoka, T. Aoki, T. Endo, A. Nukada, T. Kato and A. Hasegawa, "GPU accelerated computingfrom hype to mainstream, the rebirth of vector computing ", SciDAC 2009, Journal of Physics: Conference Series 180

[5] A. Naruse, S. Sumimotoand K. Kumon, "Acceleration Technique of Computational Fluid Dynamics on GPGPU – Over 60 GFLOPS Himeno Benchmark Performance on

1 GPU –", IPSJ SIG Technical Reports 2008-HPC-117-9 (2008)

[6] A. Naruse, Presentation at 2008 Riken Symposium, http://accc.riken.jp/HPC/Symposium/2008/naruse.pdf

[7] Mike Giles, 2008, "Jacobi iteration for a Laplace discretisation on a 3D structured grid", available at http://people.maths.ox.ac.uk/ gilesm/codes/laplace3d/laplace3d.pdf