CUDA Acceleration of Laue Depth Reconstruction Algorithm

Ke Yue, Schwarz Nicholas, and Tischler Jonathan Z.
Department of APS
Argonne National Lab
Argonne, Illinois 60439
Email: {kyue, nschwarz, tischler}@aps.anl.gov

***Abstract*—The Laue diffraction microscopy [2] experiment uses the polychromatic Laue micro-diffraction technique to examine the structure of materials with sub-micron spatial resolution in all three dimensions. During this experiment, local crystallographic orientations, orientation gradients and strains are measured as properties which will be recorded in HDF5 image format [1]. The recorded images will be processed with a depth reconstruction algorithm for future data analysis. But the current depth reconstruction algorithm consumes considerable processing time and might take up to 2 weeks for processing data collected from one single experiment. In recent years, GPU (Graphics processing units) on commodity video cards is well known for its powerful computational capability, providing an excellent platform for reducing computation cost and processing time. To improve the depth reconstruction computation speed, we propose a scalable GPU program solution on the depth reconstruction problem in this paper. The test result shows that the running time would be 25% to 30% of the prior CPU design based on the input data size.**

I. Introduction

The Laue diffraction microscopy as showed in Fig. 1 uses the polychromatic Laue micro-diffraction technique to examine the structure of materials with sub-micron spatial resolution in all three dimensions. The materials which are investigated include inter-granular and intra-granular orientation distributions in polycrystals, elastic strain tensors in elastically deformed materials, and plastic deformation microstructures under microindents in Cu single crystals. This structural microscopy techniques is very powerful for detailed investigation of the microstructure and evolution in materials, especially including local crystallographic orientations, orientation gradients and strains [2] [3]. The data set collected from the structural microscopy technique used in sector 34ID at Advanced Photon Source of Argonne National Laboratory is recorded in HDF5 format and processed by the depth reconstruction program running on CPU.

GPU is a poweful computational device for 3D graphics processing. In 2006, NVIDIA releases a new GPU architecture which facilities efficient general purpose computing on GPU (GPGPU). Unlike CPU, however, GPU has a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread quickly. GPU with massively parallel structure is more efficient than the general purpose CPU [4] for processing large blocks of data in parallel.

Work supported by U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357.

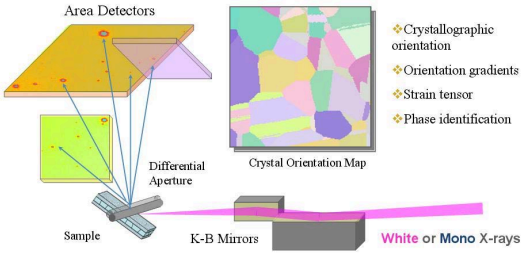

Fig. 1: Laue diffraction microscopy

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA in 2007 [5]. CUDA gives the developers access to the virtual instruction set and memory of the parallel computational elements in GPUs via C programming environment. Using CUDA, the latest NVIDIA GPUs become accessible for computation tasks like CPUs. And GPU program using CUDA has been used widely used on scientific problems such as [6], [7] and promotes the total program running speed. This paper proposes a new CUDA implementation for depth reconstruction algorithm of HDF5 images. The result shows that the new CUDA design only needs 25% to 30% of the original CPU design's running time. Section III describes the design issues for the depth reconstruction algorithm and CUDA solution to the problem. Section IV shows the experiment result and the comparison between the original CPU design and the new CUDA design.

II. Related Work

It has been mentioned in several publications that transferring significant amount of data between CPU and GPU through PCI Express link can cause a throughput bottleneck due to the limited bandwidh of PCI Express link. Schaa and Kaeli [9] admited that PCI Express will be a throughput bottleneck if a full dataset can not fit into the memory on a GPU. Owens et al. [10], Fan et al. [11], Cohen and Molemaker [12] and Dotzler et al. [13] all showed same concerns and suggest that, the communication between CPU and GPU through PCI Express should be as least as possible for GPU program.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

In order to reduce the communication cost on the running time, several strategies has been mentioned. One strategy is to overlap computation with data communication by using the memory usage pattern. Komoda et al. propose a library for OpenCL which automatically overlaps computation with data communication given the memory usage pattern of a kernel [14]. However, the programmer is required to provide various details on the memory usage pattern for the application. Another strategy is to reduce superfluous communication without reducing overlapping compassion and communication. Pai et al. propose a system that automates CPU-GPU memory management based on a coherence scheme in order to reduce superfluous communication [17]. In order to do this, the data item is transferred, only if it is not locally available on the other side, either CPU or GPU. However, this strategy can not be applied to all the scenario and specific strategy need to take for reducing superfluous communication. Furthermore, reducing superfluous may increase the computation cost and bring the total performance down.

Due to the complexities of programming for GPUs, especially communication and computation patterns for various applications, special optimization strategy should be considered for particular applications. In this paper, we provide a run-time GPU acceleration solution for depth reconstruction applications with various size of big data set. In perspective of programming pattern, we investigate the depth reconstruction applications characteristic of communication and computation patterns and try to reduce the superfluous communication for performance improvement.

III. Cuda Solution for Reconstruction Program

A. Problem

The prior design which uses CPU for the image depth reconstruction takes long time for handling big size data set. Thus we need to figure out a more efficient way for processing huge amount of data. In the following section we will propose a GPU program design which could improve the efficiency of image depth reconstruction. The input data would be a set of 2D images with pixel intensity values saved in HDF5 format.

B. Issues for Cuda Program Design

To speed up the total time on reconstructing the image, we proposed a cuda program design using GPU to handle this problem. GPU is known for its high performance capability on computation intensive applications. For designing an efficient cuda program, several challenges need to be considered in advance. The challenges are listed as the following:

- The data structure which maps each data element to a kernel thread.
- The communication time used to transfer data from CPU to GPU and vice versa needs to be minimized.
- The computation time spent on GPU for the cuda program needs to be minimized.

For handling the first challenge, the special programming characteristic of cuda program needs to be taken into consideration. The speciality of Cuda programming characteristic is that it will launch multiple kernel thread on GPU side at same time. Each kernel thread is usually doing computation on each data element in the dataset and how to map the index of the element in the dataset with the index of the kernel thread in cuda will be a challenge which needs to be considered for designing the data structures.

As mentioned in the above section, the data input is p 2D images and each image has m rows and n columns pixels as shown in Fig. 4. Each pixel in the image must be mapped to a corresponding kernel thread. In order to map each kernel thread to a corresponding pixels in the image, we could either dynamically making a 3D array or a 1D array. Then map the pixel's subscription in the 3D or 1D array to kernel thread's id (x,y,z).

To choose the more efficient data structure from 3D array or 1D array, the challenge of minimizing communication time between CPU and GPU and the computation time spent on both side needs to be considered. In aspect of communication and computation time, these two data structures can incur big performance difference, under the assumption that all the arrays are created dynamically. From the the characteristic of cuda program design, the communication time is usually spent on transferring data between CPU and GPU. The computation time is spent on the depth reconstruction of each data element and index mapping which is used to map each data element index to each thread index.

The first method is to dynamically create a 3D array, with a 1D array of pointers pointing to a 2D arrays. The advantage of this methods is that the pixel can be accessed directly based on the array subscript (x,y,z). The disadvantage is that extra pointers need to be passed from CPU side to GPU side which incur extra transferring time.

The second method is to dynamically create a 1D array, with just one pointer pointing to the first pixels of the array. The advantage of this methods is that no extra pointers are created and passed to CPU, which saves communication time. However, the disadvantage part is that the array index needs to be changed back and forth from 3D index. Extra computation time is incurred on both the CPU side and GPU side.

The difference between these two methods is that the first one needs more communication time to copy extra array pointers from CPU to GPU, while the second method needs more computation time for changing the index back and forth between 3D index and 1D index. Thus there needs a consideration between computation and communication. Communication time between CPU and GPU is usually a threshold for cuda program. Since we needs to pass extra pointers from CPU to GPU for 3D array design and GPU is known for its computation capability. It is easy to understand that 1D array would be better methods in terms of performance. Experiment is done on two different design for a 5G data set and the result is shown in Fig. 3. It could seen from the result that 1D array design could save more time on the program computation. Thus, I implement the program using the first method of creating a 1D array as the data structure used in the CUDA program.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

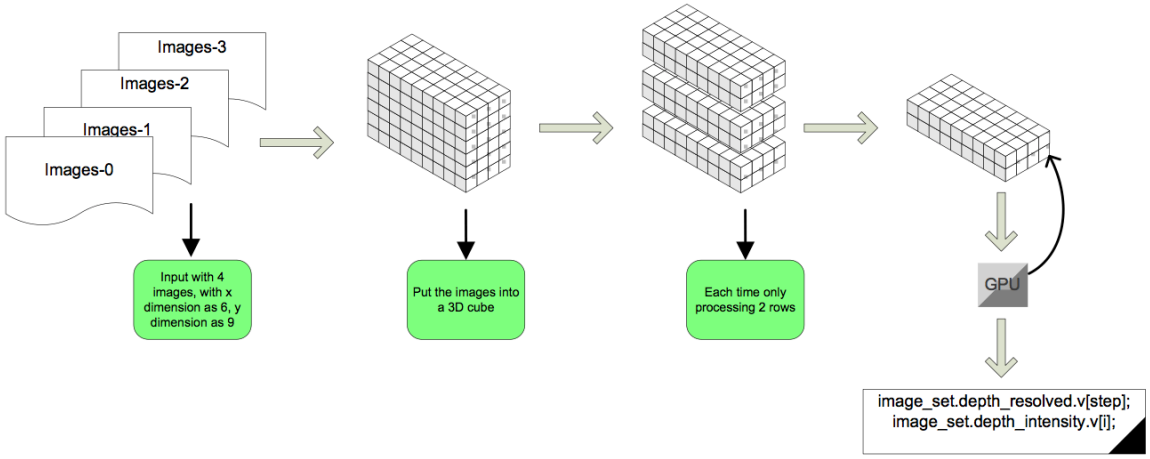

Fig. 2: Cuda program design

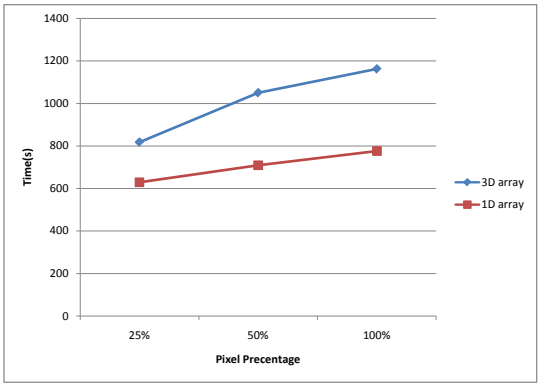

Fig. 3: Performance comparison between 1D array and 3D array implementation

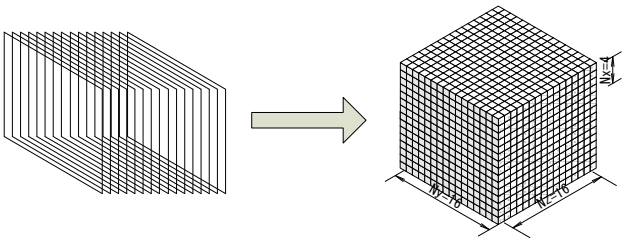

Fig. 4: Data Input

After determining the data structures, the limited memory on the video card needs to take into consideration. Since most GPU has limited memory and might not be able to handle all the data at one time. For example, the video card Tesla M2070 we used on our machine has maximum 6G memory. Other than the input data, the temporary data structures generated during the program has also to be count in. Thus, a best strategy is to divide the data input into several pieces and pass it to GPU. Take Fig. 2 as example, there are 4 images with 6 rows and 9 columns as the data input, we put all the data into a 3D data cube. And each time we only pass 2 rows into GPU side, and return the result back to CPU side. This procedure will repeat for 3 times and the final result returned to CPU side will be put it back together. In Fig. 5, detailed design is illustrated for each data set with 2 rows, 9 columns and 4 images.

C. Detail Program Design

In order to make the GPU to process the data, cuda function cudaMemcpy is used to copy the data structures designed from CPU to GPU. And after the GPU processing, the result is copied back to CPU side via cudaMemcpy again. The parameter cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost is used to specify which direction the data should be copied to.

Other than the computation part, the rest program, such as reading data from HDF5 files and writing result back to text files are still running on CPU. The start of the CUDA kernel function for computation is setTwo() function. The code slice for setTwo() function is shown in Fig. 6. After passing the data from CPU to GPU, the kernel function started to do depth reconstruction on the data input.

Taking the example shown in Fig. 2, which has 4 images with 6 rows and 9 columns as the data input, we will illustrated the cuda kernel function in detail. Each time we only pass 2 rows out of 6 rows of data to GPU side. The columns is 9 and the number of images is 4, thus the total number of pixels would 72. Therefore we launch 72 kernel threads.

As showed in Fig. 5, total number of 72 kernel threads with the dimension (2,9,4) are launched for the data set. Each kernel thread will do computation on each pixel with corresponding subscript. For example, kernel thread (1,8,1) will be mapped to data (1,8,1) in the array. Each kernel thread start with function setTwo(), which handles the thread index mapping, and call the pixel reconstruction function. The pixel reconstruction functions process the difference intensity at each pixel and then add them together. And when adding all the intensity together, an atomic cuda function is implemented for multiple

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

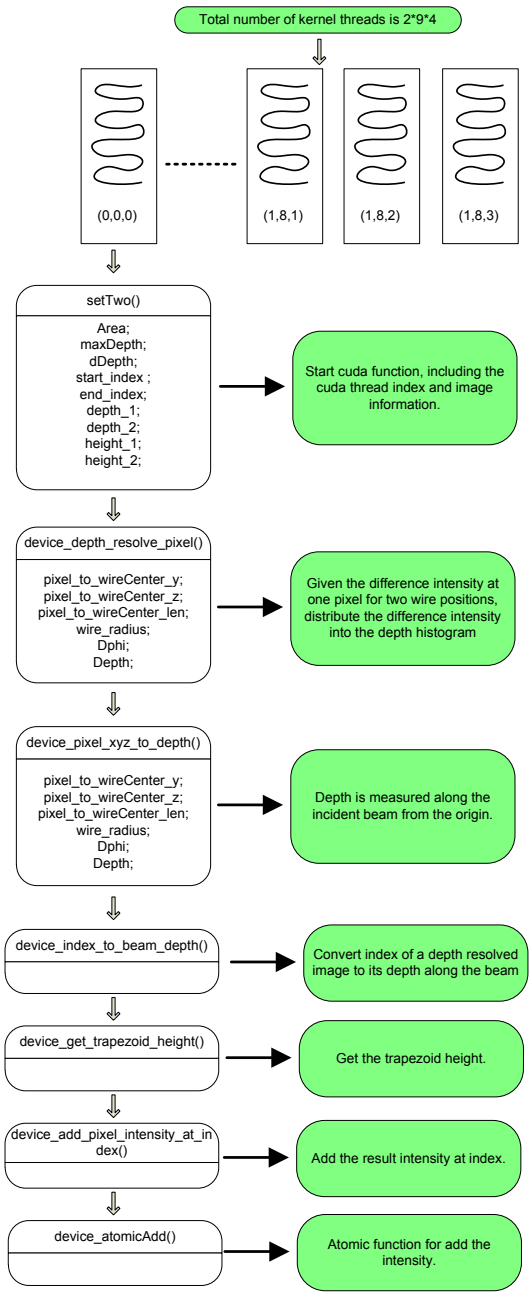

Fig. 5: Program Flow

kernel threads writing to the same arrays. Since cuda atomic functions only support integer, a function for double-precision floating-point numbers is implemented using atomicAdd().

The detailed program flow for each kernel function is illustrate in Fig. 5.

IV. Evaluation

In this section, experiments are performed on one GPU node of the cluster to compare the performance for four different data sets. It has one Nvidia Tesla M2070 GPU with 6G memory. The number of maximum threads per block is 1024

```
// device function to set the 3D volume for front edge trailing or back
edge trailing
__global__ void setTwo(double *gsl,
                       double *edge,
                       double *intensity,
                       point_xyz *firstedge,
                       int d_cutoff,
                       point_xyz *gpuPointArray,
                       double maxDepth,
                       double *image_depth,
                       double *image_depth_intensity,
                       int wire_edge,
                       int DATAXSIZE,
                       int DATAYSIZE,
                       int DATAZSIZE,
                       cudaConstPara paraPassed)
{
   unsigned idx = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned idy = blockIdx.y*blockDim.y + threadIdx.y;
   unsigned idz = blockIdx.z*blockDim.z + threadIdx.z;
   int gsl_offset = idx+idy* DATAXSIZE + DATAYSIZE * DATAXSIZE * idz;
   int instensity_offset = idx+idy* DATAXSIZE ;
   if(idy== 0)
   {
   back_edge = firstedge[idx];
   }
   else
   {
   back_edge.x = edge[idx+(idy-1)*DATAXSIZE];
   back_edge.y = edge[idx+(idy-1)*DATAXSIZE+DATAYSIZE * DATAXSIZE*1];
   back_edge.z = edge[idx+(idy-1)*DATAXSIZE+DATAYSIZE * DATAXSIZE*2];
   }
   front_edge.x = edge[idx+idy*DATAXSIZE];
   front_edge.y = edge[idx+idy*DATAXSIZE+DATAYSIZE * DATAXSIZE*1];
   front_edge.z = edge[idx+idy*DATAXSIZE+DATAYSIZE * DATAXSIZE*2];
   partial_end = device_pixel_xyz_to_depth(back_edge,
                 gpuPointArray[idz+1], wire_edge, paraPassed);
   partial_start = device_pixel_xyz_to_depth(front_edge,
                   gpuPointArray[idz], wire_edge, paraPassed);
   full_start = device_pixel_xyz_to_depth(back_edge,
                gpuPointArray[idz], wire_edge, paraPassed);
   full_end = device_pixel_xyz_to_depth(front_edge,
              gpuPointArray[idz+1], wire_edge, paraPassed);
   device_depth_resolve_pixel(diff_value, idx, idy, idz, partial_start,
              partial_end, full_start, full_end,wire_edge,image_depth,
              image_depth_intensity, DATAXSIZE, DATAYSIZE, paraPassed);
}
```

Fig. 6: Cuda functions to start the GPU kernel

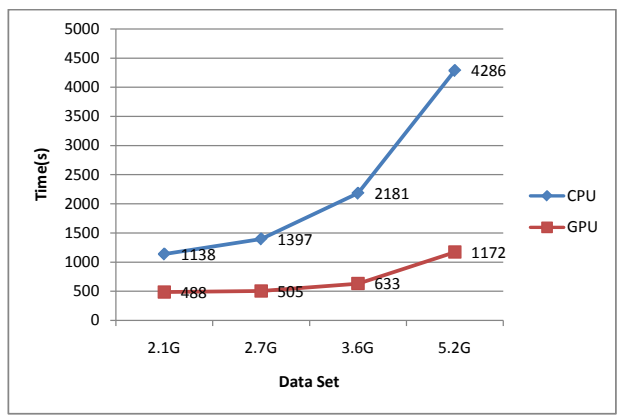

Fig. 7: Performance comparison between CPU and GPU for different data set

and the maximum block dimensions and grid dimensions is $1024 \times 1024 \times 64$, $65535 \times 65535 \times 1$ separately. The node has a 4 core CPU (Xeon E5630) running at 2.53GHz and 32G of RAM. The experiment data input is hdf5 image acquired from the detector. The following figures are the comparison result of the final running time for different data sets and pixel percentage.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

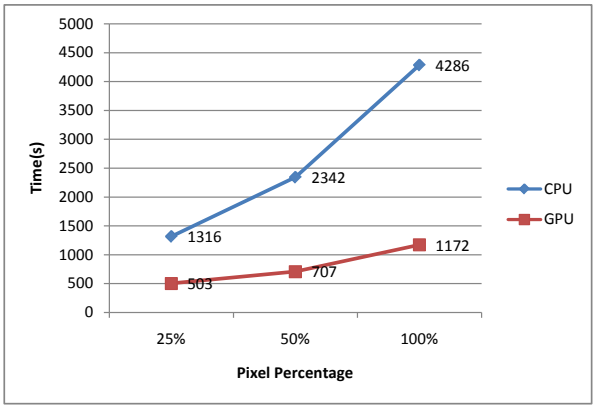

Fig. 8: Performance comparison between CPU and GPU for different pixel percentage

A. Performance Test Result

To compare the performance between CPU and GPU code, we run two experiments. The first is to change data sets from small size to big size. As showed in the Fig. 7, we have four data sets with size 2.1G, 2.7G, 3.6G, 5.2G. The total memory used for CPU and GPU code are both 4G. The final running time for CUDA design is just 25% to 30% of the original CPU version. The GPU design of the image depth reconstruction outperforms CPU version in terms of performance.

The second experiment is to change the percentage of the pixels in the data set. We sort all the pixel values and only compute certain percentage of all the pixels. We change the pixel percentage to 25%, 50%, 100%, and run two programs on the data set. The result is shown in Fig. 8 and we can conclude that the more pixels we handle, the better performance we can get. When the pixel percentage is increasing, the more data is also transferred to GPU side which incurs more data communication time. While the time saved on computation still makes the total running time for GPU code less than CPU code.

Furthermore when the data sets become bigger from 2.1G to 5.2G as showed in Fig. 7, we could see from the figure that the total running time for GPU did not change as much as CPU version. The conclusion can be got from the figure that our CUDA design did not just outperforms the CPU design in terms of performance, but also in scalability. Because GPU is usually doing very well in terms of computation intensive application. When we either changing the data size from 2.1G to 5.2G as showed in Fig. 7 or changing the pixel percentage from 25% to 100%, GPU needs to do more computation. This will enlarge the benefit that CPU can give it to the overall performance improvement, thus the explanation of the better scalability.

V. Conclusion

GPU with massively parallel structure has been proved more efficient than CPU [4] for processing large blocks of data in parallel. In this paper, we propose a GPU design methodology for the image depth reconstruction problem using CUDA. The test result shows that GPU design runs 3 or 4 times faster than the prior CPU design and thus gains a great performance improvement.

Acknowledgment

We thank Ruqing Xu from sector 34ID of Advanced Photon Source for providing HDF5 experiment data and Roger Sersted for providing IT support at Argonne National Laboratory.

References

[1] M. Folk, A. Cheng, and K. Yates, "Hdf5: A file format and i/o library for high performance computing applications," in *Proceedings of Supercomputing*, vol. 99, 1999.

[2] B. Larson, W. Yang, G. Ice, J. Budai, and J. Tischler, "Three-dimensional x-ray structural microscopy with submicrometre resolution," *Nature*, vol. 415, no. 6874, pp. 887–890, 2002.

[3] W. Liu, P. Zschack, J. Tischler, G. Ice, and B. Larson, "X-ray laue diffraction microscopy in 3d at the advanced photon source," in *THE 10TH INTERNATIONAL CONFERENCE ON X-RAY MICROSCOPY*, vol. 1365, no. 1. AIP Publishing, 2011, pp. 108–111.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *Journal of parallel and distributed computing*, vol. 68, no. 10, pp. 1370–1380, 2008.

[5] C. Nvidia, "Programming guide," 2008.

[6] E. Shalaby and M. A. Oliveira, "Using gpus to compute fast fourier transforms for crystal structure solution and refinement," *Journal of Applied Crystallography*, vol. 46, no. 3, pp. 594–600, 2013.

[7] P. Hobson, J. Nebrensky, and I. Reid, "Challenges in using gpus for the real-time reconstruction of digital hologram images," in *Journal of Physics: Conference Series*, vol. 415, no. 1. IOP Publishing, 2013, p. 012042.

[8] J. Choe, A. Nikoozadeh, Ö. Oralkan, and B. Khuri-Yakub, "Gpu-based real-time volumetric ultrasound image reconstruction for a ring array," 2013.

[9] D. Schaa and D. Kaeli, "Exploring the multiple-gpu design space," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[10] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[11] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004, p. 47.

[12] J. Cohen and M. J. Molemaker, "A fast double precision cfd code using cuda," *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, pp. 414–429, 2009.

[13] G. Dotzler, R. Veldema, and M. Klemm, "Jcudamp: Openmp/java on cuda," in *Proceedings of the 3rd International Workshop on Multicore Software Engineering*. ACM, 2010, pp. 10–17.

[14] T. Komoda, S. Miwa, and H. Nakamura, "Communication library to overlap computation and communication for opencl application," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 567–573.

[15] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for cpu-gpu architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 165–174.

[16] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communication management and optimization," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 142–151.

[17] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 33–42.