Accelerating Particle Image Velocimetry Using Hybrid Architectures

Vivek Venugopal *, Cameron D. Patterson *, Kevin Shinpaugh †
*Bradley Department of Electrical and Computer Engineering
† Advanced Research Computing
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061
Email: vivekv@vt.edu, cdp@vt.edu, kashin@vt.edu

***Abstract*—High Performance Computing (HPC) applications are mapped to a cluster of multi-core processors communicating using high speed interconnects. More computational power is harnessed with the addition of hardware accelerators such as Graphics Processing Unit (GPU) cards and Field Programmable Gate Arrays (FPGAs). Particle Image Velocimetry (PIV) is an embarrassingly parallel application that can benefit from acceleration using hybrid architectures. The PIV application is mapped to a Nvidia GPU system, resulting in 3x speedup over a dual quad-core Intel processor implementation. The design methodology used to implement the PIV application on a specialized FPGA platform under development is described in brief and the resulting performance benefit is analyzed.**

I. Particle Image Velocimetry Algorithm

Cardiovascular Disease (CVD) is the leading cause of death in the United States and accounts for more than 36.3 % of all fatalities for 2006 [1]. Early and accurate detection of CVD would increase the median human life by a significant percentage. To facilitate this detection, the Advanced Experimental Thermofluids Engineering (AEThER) Lab at Virginia Tech is involved in the area of cardiovascular fluid dynamics. Their research focuses on the effects of coronary stents on blood flow, and the role of pressure gradients in the development of diastolic dysfunction and heart failure. The experimental method [2] involves using the data intensive Particle Image Velocimetry (PIV) technique for flow measurement and visualization. The PIV technique utilizes a high intensity pulsed laser sheet to illuminate a flow field seeded with reflective or fluorescent particles. The motion of the particles are captured using high speed digital cameras that produce images of the particle patterns. These images are correlated using the PIV algorithm to measure the fluid velocity, which influences the stress exerted on the arterial walls. The experimental setup produces large amount of data (each case results in 1250 image pairs $\times$ 5MB = 6.25 GB). On an average, there are 100 cases generated for each stent configuration from a total of 20 stent configurations (about 500 GB of data for each case).

The PIV algorithm shown in Figure 1 consists of interrogation windows to capture the flow field. The window sizes are 16$\times$16, 32$\times$32 or 64$\times$64 pixels. Each image is broken down into overlapping zones and the corresponding zones from both images are correlated. The peak detection is done

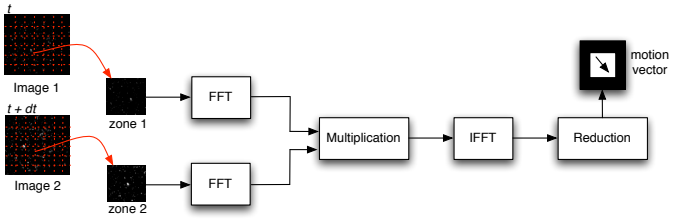

Fig. 1. The PIV algorithm

using the Fast Fourier Transfor (FFT) block, which is then translated to depict the direction of the particles within the zone. The reduction block consists of a sub-pixel algorithm and a filtering subroutine to compute the velocity value. The complete algorithm was initially coded by the AEThER lab using their native FlowIQ program. The FlowIQ program analyses each image pair in 16 minutes on a 2GHz Xeon processor. This would take about 2.6 years for the complete dataset without accounting for image pre-processing and post-processing analysis.

We implement the PIV algorithm in C with the FFT routines ported using the fast FFTW library [3]. The sequential C program is executed on Virginia Tech's System G supercomputer and provides a benchmark for comparison with implementation on GPGPU and FPGA platforms. Section II describes the Compute Unified Device Architecture (CUDA) programming model used in the GPU architecture. Section III describes the PRO-PART design flow for FPGA implementation in brief. Section IV presents the experimental results and the concluding remarks are given in section V.

II. Compute Unified Device Architecture

General Purpose computation on Graphics Processing Units (GPGPU) is enhanced by the Nvidia's Compute Unified Device Architecture (CUDA), where the GPU is used as a multi-core co-processor in addition to graphics rendering. CUDA presents a heterogeneous programming model, where the GPU can be used in conjunction with the CPU for parallel execution. The Nvidia Tesla 10-series architecture [4] consists of 30 streaming multi-processors with 8 cores each as shown in Figure 2. All the 8 streaming processors have common access to 16 KB of shared memory within a multi-processor. Each

multiprocessor has one set of 32-bit registers per processor, constant memory and texture caches. Each streaming core can execute the same instruction on different data making it similar to a Single Instruction Multiple Data (SIMD) processor. The multi-processors communicate with the CPU through the GPU memory using the PCI Express interface.

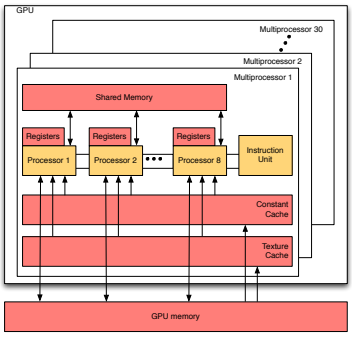

Fig. 2. CUDA hardware model for Tesla 10-series architecture

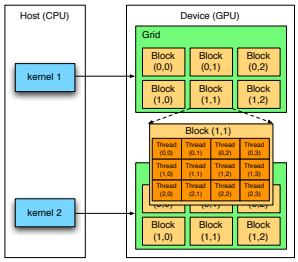

Fig. 3. CUDA programming model

From a programmer's point of view, the CUDA model in Figure 3 represents the GPU as a multi-threaded parallel architecture. The CPU is referred to as the host and the GPU is referred to as the device. CUDA assumes that the host and the device have separate accesses to their memory, also referred to as host memory and device memory. The CUDA execution is based on *threads* and a collection of threads is called a *block*. A group of blocks can be assigned to a single multi-processor and time-share their execution. A *grid* constitutes of a collection of blocks in a single execution. Each thread and block can be accessed within the thread using a unique identifier. The *kernel* is the code executed on each thread. Using the thread and block identifier, the thread performs the kernel task on its part of the data. Multiple kernels can be called by an algorithm and the kernels share data through the global memory. Synchronization barrier is available between threads executing inside a block. However, there is no synchronization between the blocks executing within a kernel. The threads from multiple blocks are synchronized after the end of the kernel execution. The CUDA API consists of a set of library functions, which are extensions of the C language. The *nvcc* compiler generates executable code for the CUDA device. The CUDA compiler uses the FFTW library for FFT computation.

III. PRO-PART Design Flow

A multi-core Streaming Architecture without Flow Control (SAFC) testbed is developed using the Xilinx ML310 Embedded Development board consisting of a Virtex-II Pro XC2VP30 FPGA. Each ML310 is configured with a mesh of PEs and the on-board communication between the PEs is handled using the Xilinx FSL interface [5]. The FSL interface is an example of point-to-point interconnect and suits the research focus of this work. The ML310 boards are physically connected by InfiniBand cables in a mesh topology. The off-board communication is facilitated by using Xilinx's simple and lightweight Aurora protocol [6]. This specialized multi-core platform is called NOFIS, because the platform consists of a **N**etwork **O**f **F**PGAs with **I**ntegrated Aurora **S**witches, as shown in Figure 4.

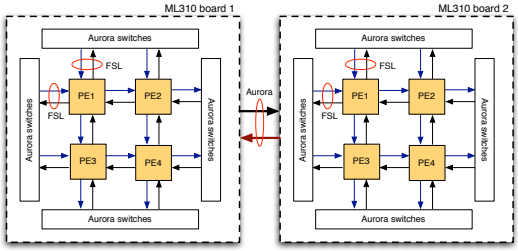

Fig. 4. Multi-core SAFC hardware: NOFIS platform

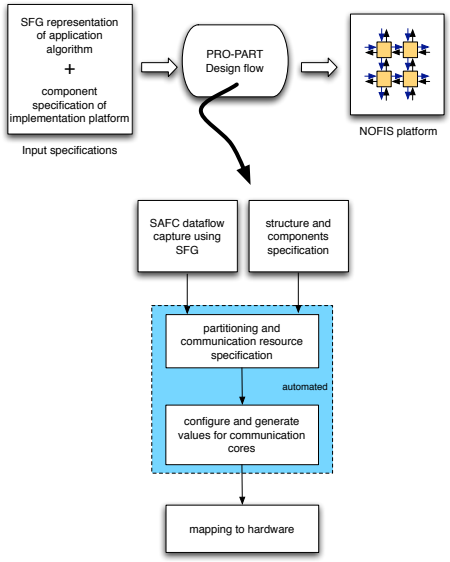

Fig. 5. PRO-PART design flow

The design methodology provides a formalism to describe the synchronization problems in streaming dataflow architectures. The proposed methodology is referred to as PRO-PART design flow, due to the significant **pro**cess **part**itioning stage. The PRO-PART design flow shown in Figure 5 takes the SAFC

Flow Graph and platform specification as inputs. The PRO-PART design flow automates partitioning of the processes and scheduling of the communication resources. Also, the design flow generates values for the customizable communication cores without iterated redesign. The embedded flow control and the communication resource parameters are mapped on the hardware platform.

IV. Experimental Results

System G consists of 324 Apple Mac Pro nodes with 2 quad-core Intel Xeon processors and 8GB of RAM per node running a 64-bit version of Linux CentOS. The CPU benchmark was executed on a single node using C and the fast FFTW library. The CUDA benchmarks were obtained using Nvidia's Tesla C1060 GPU installed on a System G node. The Tesla C1060 GPU consists of 240 streaming processor cores clocked at 1.3 GHz with 4 GB onboard memory and utilizes a PCI Express x16 slot. The sequential C PIV program consists of 15 functions called repeatedly to find the 3096 indexes, indicating the flow displacement in the XY direction. Each function performs data processing on 4096 elements representing the 64×64 window size. The CUDA programming model supports thread-based parallel processing, where each function in C is represented by a kernel executed N times in parallel by N different threads. By using the CUDA approach, each kernel computes the 4096 elements for 3096 positions in parallel.

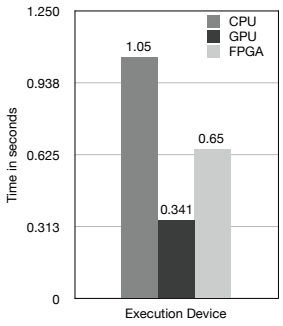

Fig. 6. Execution time comparison of the three platforms

The PIV application exhibits linear speedup to the number of cores available for execution. Figure 6 shows the execution time required for computing all the 3096 indices of the PIV algorithm. The NVIDIA CUDA 2.2 API with *memcopy* implementation exhibits a speedup of 3x over the sequential C program. The performance of the PIV algorithm on the NOFIS platform is estimated using the number of clock cycles required for each of the custom computational blocks, the latency of the Aurora interface and the number of instructions executed per clock cycle.

The bottleneck in the CUDA implementation occurs when data has to be copied from device memory to host memory or vice-versa. Figure 7 shows the comparison of the GPU execution time for three different configurations. The PIV application is implemented with the *memcopy* function call using the CUDA 2.1 and CUDA 2.2 APIs. The CUDA 2.2 API introduces new function calls that allow host memory to be allocated and mapped to device memory using pinned memory buffers (*zerocopy* feature). The CUDA 2.2 API implementation with the *zerocopy* feature exhibits maximum processing time on the GPU. This may be due to the discrete GPU card, as the memory is not cached by the GPU.

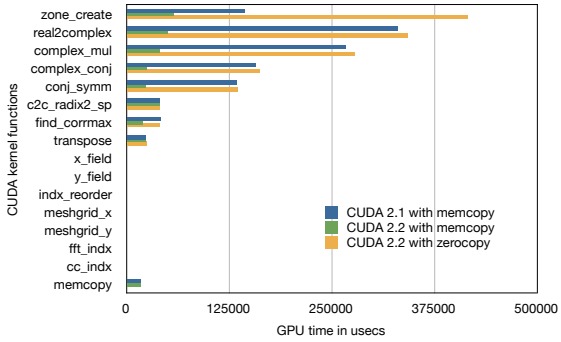

Fig. 7. CUDA profile graph comparison for three different configurations

V. Conclusion

Nvidia's Tesla C1060 GPU provides computational speedup as compared to both the sequential CPU implementation and the NOFIS implementation for the PIV application. The synchronization and the latency in data movement can be optimized between the FPGAs by having custom communication interfaces without an Operating System (OS) overhead. The PRO-PART design flow facilitates a fast and easier mapping of a streaming application on the specialized NOFIS platform with a significant advantage in development time.

Acknowledgment

The authors would like to thank Dr. Pavlos P. Vlachos, John Charonko and Adric Eckstein of the AEThER lab at Virginia Tech for their guidance on the PIV technique. Many thanks are also due to Dr. Srinidhi Varadarajan for providing access to System G. The Nvidia Tesla C1060 GPU was funded through the Nvidia Professor Partnership program.

References

[1] American Heart Association. (Last Accessed: February 2009) Cardiovascular Disease Statistics. [Online]. Available: http://www.americanheart.org/presenter.jhtml?identifier=4478

[2] A. Eckstein, J. Charonko, and P. Vlachos, "Phase correlation processing for DPIV measurements," *Experiments in Fluids*, 2008. [Online]. Available: http://dx.doi.org/10.1007/s00348-008-0492-6

[3] M. Frigo and S. G. Johnson, *FFTW3.2 manual*, 2007.

[4] Nvidia Inc. (Last Accessed: February 2008) Nvidia Tesla C1060 GPU Computing Processor. [Online]. Available: {http://www.nvidia.com/object/product_tesla_c1060_us.html}

[5] Xilinx Inc., *Fast Simplex Link (FSL) Bus (v2.11a) manual*, Last Accessed: June 2008. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pd%f

[6] Xilinx Inc., *LogiCORE Aurora v2.8*, Last Accessed: June 2008. [Online]. Available: http://www.xilinx.com/aurora/auroramember/ug061.pdf