GPU-Based Parallel Computing: A New Computational Approach and its Applications to Nuclear Engineering

Kai Huang[1], Volodymyr Kindratenko[2], Rizwan-uddin[1, 2]

1. *Department of Nuclear, Plasma, & Radiological Engineering, University of Illinois at Urbana-Champaign*
104 South Wright Street, Urbana, IL 61801, USA
2. *National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign*
1205 West Clark Street, Urbana, IL 61801, USA
khuang21@illinois.edu; kindrtnk@illinois.edu; rizwan@illinois.edu

INTRODUCTION

Faster speed and higher accuracy are, and will always be pursued by computational scientists and engineers. To achieve these goals, single-core micro-processors were assembled to arrive at "parallel computing". This parallelism is recently extended to chip-level with the emergence of multi- and many-core architectures—that is, roughly, adding more cores onto a single chip—in both Central Processing Units (CPUs) and Graphics Processing Units (GPUs). As a result, scientific computing is rapidly moving toward many-core parallelism. Modeling and simulation in nuclear science and engineering relies heavily on computational power. Nuclear engineers took advantage of parallelization to advance the simulations of neutronics, thermal hydraulics, materials, plasma physics, etc [2, 3]. The field is expected to benefit significantly from the new many-core GPU technology.

Although the current mainstream CPUs and GPUs have both been multi-cored, the latter is lately realized to be a possibly stronger candidate than the former to solve computationally demanding problems. This is due to the fact that the floating-point computational capability of GPUs becomes generally 10× higher than CPUs, which have dominated the computational realm over the last few decades. A GPU has many more transistors devoted to data processing than caching and controlling [4]. The impressive computing performance of GPUs today originates from their streamlined functionality and design.

Up until recently, GPUs were used exclusively in computer graphics. Their use for general-purpose (GP) computations was very limited both due to the fixed hardware that was designed for one specific purpose and due to the difficulties in programming using graphics-oriented languages, such as OpenGL. The situation has changed dramatically with the recent introduction of GPU architectures by NVIDIA and ATI/AMD that are fully programmable and with the introduction of programming models and tools, such as NVIDIA's Compute Unified Device Architecture (CUDA) that greatly simplified the development of applications for GPUs. Since 2007, many applications have been re-written in CUDA to run on GPUs yielding very substantial performance benefits.

Figure 1. shows the architecture of NVIDIA Quadro 5600 GPU, one of GPUs from NVIDIA that are primarily used not as a graphics processor, but as a compute co-processor. The basic processing unit of NVIDIA GPU is the streaming processor (SP), a fully pipelined, single-issue, in-order microprocessor complete with two arithmetic logic units (ALUs) and a floating-point unit (FPU). A group of 8 SPs, 2 additional special function units (SFUs)—processor cores that have floating-point multiply units used for transcendental operations—and 16 KB of shared memory form a streaming multiprocessor (SM). A group of 2 SMs with some additional memory form texture/processor cluster (TPC). Sixteen such clusters form the streaming processor array. In total, Quadro 5600 GPU has 128 SPs running at 1.35 GHz, thus delivering 346 GFLOPs in single-precision floating-point arithmetic. 384-bit interface to the off-chip GDDR3 memory provides 76.8 GB/sec bandwidth.

CUDA is a programming platform that programmers can use to realize algorithms on GPUs. Although several other programming environments, such as Brook, Sh, and CTM are also available for GPU computing [5], CUDA is the most popular. Developed by NVIDIA, CUDA makes it for the first time possible to manipulate the GPU hardware and exploit its tremendous potential in science and engineering computing without knowing the details of graphics libraries [6].

CUDA exposes the GPU hardware as a number of MIMD (Multiple Instruction, Multiple Data) multi-processors containing a set of SIMD (Single Instruction, Multiple Data) processors [4]. A global memory is reachable for all multiprocessors, while a shared memory is reachable for all

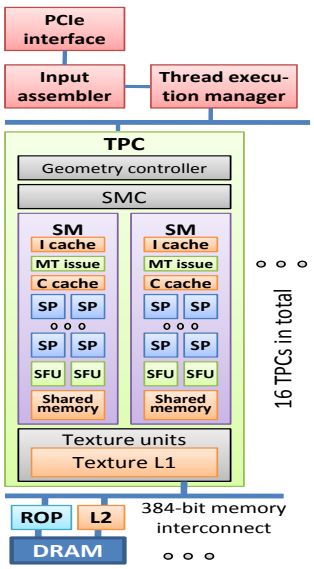

Fig. 1. NVIDIA Quadro 5600 GPU architecture.

processors within a multiprocessor. The basic executing unit in a CUDA program is a thread. Each thread executes exactly the same kernel function. Threads are distinguished by the identifiers automatically assigned to each of them by the system. Threads are also associated with several registers to store local variables. A number of threads are grouped to form a thread block. Synchronization among threads is achievable only within a block.

The execution of a CUDA program starts from the host side where CPU executes the serial part of the code. Once the kernel function is called, CPU invokes GPU on the device side and a large number of threads (many thousands) are loaded to run the code of the kernel function simultaneously.

AN EXAMPLE: GPU IMPLEMENTATION OF THE 2-D BURGERS EQUATION

Burgers equation is a non-linear model of the convection-diffusion process similar to the Navier-Stokes equations, but without the pressure term. It has been used to test new numerical schemes [7] and new computer architectures prior to their applications to, for example, the Navier-Stokes equations.

The 2-D Burgers equation is implemented on an NVIDIA Quadro 5600 GPU. Explicit Euler and second order central difference schemes are applied to discretize time and space, respectively. To parallelize, an individual thread created on the GPU is assigned to each finite difference grid point in the computational domain. Hundreds of GPU threads work simultaneously to update the unknown variables at each time step. Hence the update process, namely, calling the kernel function, is executed with the time evolution without sweeping the space domain at each single step, which is a compute demanding part in a serial implementation. Pseudo-code for the CUDA implementation is given in Appendix A [8], and the burgers kernel function is shown in Appendix B.

Numerical solution is obtained over the unit square, $0 \le x \le 1, 0 \le y \le 1$ with homogenous initial conditions and the following boundary conditions [9]

$u(0,y) = u(1,y) = v(x,1) = 0, v(x,0) = 1,$

$u(x,0) = u(x,1) = sin2\pi x, v(0,y) = v(1,y) = 1 - y.$

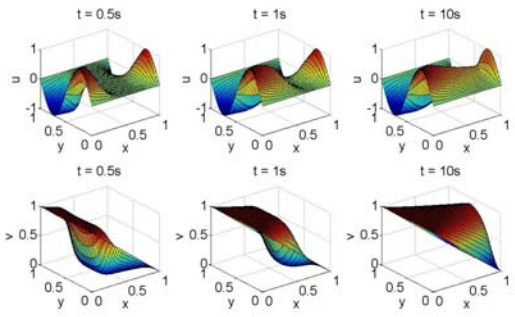

Fig. 2. Snapshots showing *u* and *v* velocity components.

Steady state solution can be achieved by marching in time. To make the problem more compute demanding, a very small time step (Δt=10^{-5}s) is chosen. The time step affects both CPU and GPU based simulations the same way. Hence, its only effect is to increase the compute time for both; allowing better ratio comparison. The GPU implementation of the problem is coded in CUDA. A CPU serial implementation of the same problem is coded in C for comparison purposes. The C code runs on a PC which has an Intel Core2 Duo CPU running at 2.50GHz. Numerical results for *u* and *v* velocities at different times are shown in Fig. 2. Table I shows performance comparison of the CPU and GPU implementation for 2-D Burgers equation. A speedup of two to five times is obtained for GPU for meshes from 22×22 to 46×46. It is observed that better speedup is obtained for larger size problems. As is well known from parallel computing exercises, this trend is expected to continue for even larger size problems.

TABLE I. Performance comparison of CPU and GPU implementation for 2-D Burgers equation

mesh	GPU time (s)	CPU time (s)	overall speedup
22×22	14.49	30.91	2.13
32×32	18.94	61.67	3.26
46×46	28.82	147.88	5.13

APPENDIX A: PSEUDO-CODE FOR 2-D BURGERS SOLVER ON CUDA

1) Initialization on host (CPU)
   - Dynamic memory allocation on host
   - Set initial and boundary conditions
   - Calculate constant coefficients
2) Initialization on device (GPU)
   - Set up device execution configuration
   - Dynamic memory allocation on device
3) Copy variables from host memory to device memory
4) Time marching loops: call burgers kernel in each time step
5) Copy results from device memory back to host memory
6) Write results into output files

APPENDIX B: BURGERS KERNEL FUNCTION

```
__global__ void burgers_kernel<<<dimGrid,
dimBlock>>>(float* u, float* v, float* u_old, float* v_old)
{
// Block indices
int bx = blockIdx.x;
int by = blockIdx.y;
```

```
// Thread indices
int tx = threadIdx.x;
int ty = threadIdx.y;
// Global thread indices
int global_tx = bx * BLOCK_SIZE + tx;
int global_ty = by * BLOCK_SIZE + ty;

// Mapping from thread indices to mesh indices
int row = global_ty + 1;
int col = global_tx + 1;

int k = row * NUM_GRIDS_X + col;

// Explicit Euler update rules
u[k] = u_old[k] + dt * (Mu * ((u_old[k+NUM_GRIDS_X]
- 2 * u_old[k] + u_old[k-NUM_GRIDS_X]) / (dx * dx) +
(u_old[k+1] - 2 * u_old[k] + u_old[k-NUM_GRIDS_X]) /
(dy*dy)) - u_old[k] * (u_old[k+NUM_GRIDS_X] -
u_old[k-NUM_GRIDS_X]) / (2 * dx) - v_old[k] *
(u_old[k+1] - u_old[k-1]) / (2 * dy));

v[k] = v_old[k] + dt * (Mu * ((v_old[k+NUM_GRIDS_X]
- 2 * v_old[k] + v_old[k-NUM_GRIDS_X]) / (dx*dx) +
(v_old[k+1] - 2 * v_old[k] + v_old[k-NUM_GRIDS_X]) /
(dy*dy)) - u_old[k] * (v_old[k+NUM_GRIDS_X] -
v_old[k-NUM_GRIDS_X]) / (2 * dx) - v_old[k] *
(v_old[k+1] - v_old[k-1]) / (2*dy));

}
```


ACKNOWLEDGEMENT

This work was partially supported by the NCSA/ UIUC Faculty Fellows Program, and utilized the NCSA Accelerator Cluster [10].

REFERENCES

1. J. D. OWENS, M. HOUSTON, D. LUEBKE, S. GREEN, J. E. STONE, and J. C. PHILLIP, "GPU Computing," *Proc. IEEE,* **96(5)**, 879-899 (2008).
2. Y. Y. AZMY, "Performance and Performance Modeling of a Parallel Algorithm for Solving the Neutron Transport Equation," *J. Supercomp.*, **6**, 211 (1992).
3. Y. Y. AZMY, "Multiprocessing for Neutron Diffusion and Deterministic Transport Methods," *Prog. Nucl. Energy*, **31**, 317 (1997).
4. NVIDIA, C. (2008). *NVIDIA CUDA Programming Guide (version 2.0)*. NVIDIA Corporation.
5. E. ELSEN, P. LEGRESLEY, E. DARVE, "Large Calculation of the flow over a hypersonic vehicle using a GPU," *J. Comp. Phys.*, **227**, 10148-10161 (2008).
6. A. J. RUEDA, L. ORTEGAM, "Geometric Algorithms on CUDA," [online] Available: http://www.nvidia.com/ docs/IO/47905/cuda-grapp.pdf
7. B. WESCOTT and RIZWAN-UDDIN, "An Efficient Formulation of the Modified Nodal Integral Method and Application to the Two-Dimensional Burgers' Equation," *Nucl. Sci. Eng.*, **139**, 293-305 (2001).
8. A. F. SHINN, "Computational Fluid Dynamics (CFD) using Graphics Processing Units," [online] Available: http://www.greatlakesconsortium.org/events/GPUMulticore/Shinn_talk.pdf
9. P. MOIN, *Fundamentals of Engineering Numerical Analysis*, p. 149, Cambridge University Press (2001).
10. M. SHOWERMAN, J. ENOS, A. PANT, V. KINDRATENKO, C. STEFFEN, R. PENNINGTON, W. HWU, "QP: A Heterogeneous Multi-Accelerator Cluster," Proc. *10th LCI International Conference on High-Performance Clustered Computing*, 2009.