Parallel Computing for Long-Time Simulations of Calcium Waves in a Heart Cell

Yu Wang[1], **Marc Olano**[1], **Matthias Gobbert**[2,*], and **Wesley Griffin**[1]

[1] Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County
[2] Department of Mathematics and Statistics, University of Maryland, Baltimore County

Calcium waves are modeled by parabolic partial differential equations, whose simulation codes contain Krylov subspace methods as computational kernels. This paper presents GPU-based parallel computations for the conjugate gradient method applied to the finite difference discretization of a Poisson equation as prototype problem for the computational kernel. The CUDA algorithm tests the three memory systems of global memory, texture memory, and shared memory of a CUDA-enabled GPU. Due to the caching mechanism and coalesced read/write operations, the CUDA algorithm using global memory and single precision floating point numbers outperforms algorithms accessing texture memory and the shared memory.

© 2012 Wiley-VCH Verlag GmbH & Co. KGaA, Weinheim

1 Introduction

The Poisson equation with homogeneous Dirichlet boundary condition

$$
\begin{aligned}
-\Delta u &= f \quad \text{in } \Omega, \\
u &= 0 \quad \text{on } \partial\Omega.
\end{aligned}
\tag{1}
$$

on a unit square domain in two dimensions $\Omega = (0,1) \times (0,1) \subset R^2$ be numerically approximated with the finite difference method, which results in a system of linear equations for the approximations at the mesh points. This is a classical test problem for linear solvers, in particular in parallel computing, due to the problem structure that is typical for many numerical methods for partial differential equations. Krylov subspace methods including the conjugate gradient method are popular solvers for this type of system as well as are important as computational kernels inside of time-dependent parabolic simulation codes, since the system matrix only enters the algorithm via matrix-vector products, which in turn can be implemented directly without storing the matrix, a so-called matrix-free implementation of the method. The parallelism of Graphics Processing Units (GPUs) offers the opportunity for excellent performance for problems of this type due to their regular structure. We present here the results of studies on an NVIDIA Tesla C2070 that compares three strategies of memory access to results using a serial CPU.

2 CUDA Memory Model

CUDA was introduced in 2007 by NVIDIA to enable programming general purpose computation on parallel GPU architectures [1]. (We refer the reader to [2] for more references to previous work using the CUDA API for general purpose computation.) Global memory is GPU memory visible to every thread in the same compute grid. Shared memory is visible to threads in the same compute block and is very fast to access, but has much smaller capacity than global memory. Texture memory shares space with global memory, including caching in 2D blocks and the option to use specialized linear blending hardware.

3 CUDA Algorithms

We have designed three different CUDA algorithms to solve this linear system on the GPU. All three store the large vector data in global memory, but they differ in how the data are accessed each iteration. Global memory is a large DRAM store with long access latency (hundreds of cycles) and limited access bandwidth. There are three distinct strategies for accessing global memory that differ in how they hide or amortize the access latency. Our first algorithm, which we call the *direct global memory* solution, uses *coalesced* data reads with repeated reads for each access to the same data. This solution relies on the latency hiding of the coalesced read, and the fast L1 caching to reduce the data access time. The second version, which we will call the *texture memory* solution, organizes the vector data to enable access using texture operations. This solution relies on the 2D block texture caching and optimized texture fetch hardware to reduce access time. The final version, which we call the *shared memory* solution, organizes the data to allow data sharing between threads in shared memory. Data is explicitly fetched from global to shared memory according to the known access patterns.

---

* Corresponding author: e-mail gobbert@umbc.edu, phone +1 410 455 2404, fax +1 410 455 1066

The Direct Global Memory algorithm explicitly fetches each sample in the 5-sample cross pattern as a global memory read. It relies on local caching to manage data reuse. In addition, in order to achieve better global memory access efficiency, parallel threads in a warp read data from consecutive memory addresses to form a coalesced read (when fetching data in the same cache line, the active threads are grouped into one consolidated operation). The 2D texture memory solution is able to take advantage of texture hardware boundary support, which eliminates the logic checking for boundary conditions. In addition, 2D texture memory caches data elements in blocks, which should match well to the access pattern in our test problem. Shared memory is located close to the processors for each thread block, with similar performance to registers. Due to capacity limits, data blocks must be paged on and off the shared memory in a coalesced manner. Shared memory is only visible to threads within the same block; threads in one thread block cannot access the shared memory in a different block. This characteristic guarantees the independence of kernel execution and memory update in each individual thread block. Communication between threads in different blocks must use global memory, and any synchronization is a blocking operation. To avoid inter-block communication, we duplicate data in a margin around each block. The shared memory code 1) copies square data blocks from the global memory to the shared memory in separate thread blocks. Ghost regions, i.e., the data adjacent to the boundaries of the square data block, are copied by separate threads. This data is copied from the global memory into four arrays located in the shared memory of each thread block; 2) perform cross-pattern calculation on the elements and get partial results, using elements from the ghost regions as needed; 3) copy the square data blocks from the shared memory back to global memory.

4   Results

Since we are merely changing the time complexity of the matrix-vector multiplication in the conjugate gradient method, the numerical results of the various algorithms we have discussed in the paper should converge at the same rate with the same numerical results. Our experiments verify this to be true [2]. According to our experimental results, when using double precision floating point numbers, direct global memory algrithm converges at exactly the same rate as the shared memory algorithm. We perform experiments using floating point numbers with both single and double precision, thus the numbers of iterations in the CUDA algorithms differ slightly for the same tolerance of 10^{-6} for the conjugate gradient method. Figure 1 shows the average execution time of one iteration for each algorithm. The experimental results for the serial code were collected using a single core on a quad core Intel Nehalem X5550 processor. The speedup of our CUDA algorithms over the serial run is presented in Figure 2.
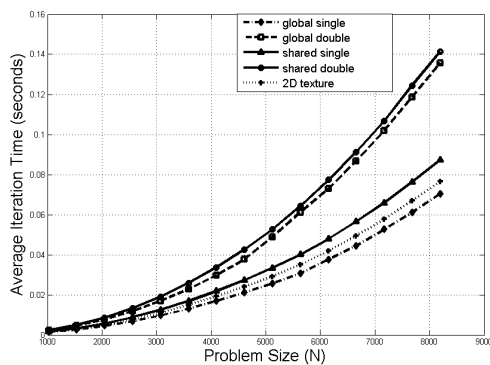

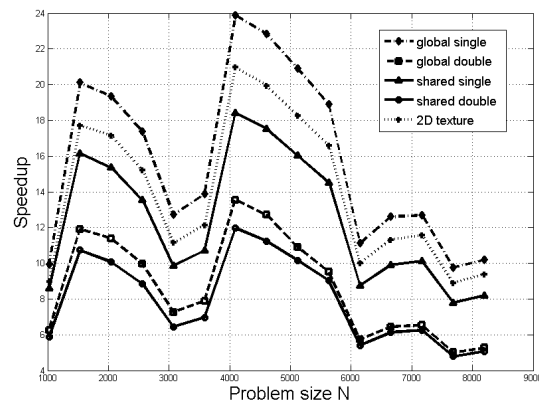**Fig. 1:** Average Iteration Time

Fig. 2: Speedup over serial code

5   Conclusion

We have presented GPU-based solutions to the elliptic test problem of a Poisson equation with Dirichlet boundary conditions in two spatial dimensions. Using a matrix-free implementation of the iterative Conjugate Gradient method relieves some memory concern, which is the major performance constraint on modern GPUs. From the experimental results in the convergence study, coalesced direct access to global memory, even with repetitive accesses to the same global memory locations, performs best on this type of problem. Next best is using 2D texture memory access, though this limits the computation to single precision. Due to the caching mechanism of global memory, the shared memory solution is better suited for larger problem sizes.

References

[1] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 4.2, April 2012.
[2] Y. Wang, M. Olano, M. K. Gobbert, and W. Griffin, A GPU memory system comparison for an elliptic test problem, Tech. Rep. HPCF–2012–1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2012.