# Heterogeneous Computing in Economics: A Simplified Approach

Matt P. Dziubinski and Stefano Grassi

CREATES Research Paper 2012-15

# HETEROGENEOUS COMPUTING IN ECONOMICS: A SIMPLIFIED APPROACH

MATT P. DZIUBINSKI AND STEFANO GRASSI

*CREATES, Department of Economics and Business, Aarhus University, Denmark*

ABSTRACT. This paper shows the potential of heterogeneous computing in solving dynamic equilibrium models in economics. We illustrate the power and simplicity of C++ Accelerated Massive Parallelism (C++ AMP) recently introduced by Microsoft. Starting from the same exercise as Aldrich et al. (2011) we document a speed gain together with a simplified programming style that naturally enables parallelization.

*JEL Classification*: C88.

*Keywords*: Code optimization; CUDA; C++; C++ AMP; Data parallelism; DSGE models; Econometrics; Heterogeneous computing; High-performance computing; Parallel computing.

## 1. INTRODUCTION

This paper shows the potential of heterogeneous computing in solving dynamic equilibrium models in economics. Heterogeneous computing refers to the use of different processing cores (types of computational units) to maximize performance. [1]

We rely on C++ Accelerated Massive Parallelism (C++ AMP), a new technology introduced by Microsoft, that allows to use accelerators, such as graphics processing units (GPUs), to speed up calculations. GPUs are a standard part of the current personal computers and are designed for data

*E-mail address*: matt@creates.au.dk.

*Date*: June 17, 2012.

We acknowledge financial support by the Center for Research in Econometric Analysis of Time Series, CREATES, funded by the Danish National Research Foundation.

[1]"A computational unit could be a general-purpose processor (GPP) – including but not limited to a multi-core central processing unit (CPU), a special-purpose processor (i.e. digital signal processor (DSP) or graphics processing unit (GPU), a co-processor, or custom acceleration logic (application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA)). In general, a heterogeneous computing platform consists of processors with different instruction set architectures (ISAs)." Source: `http://en.wikipedia.org/wiki/Heterogeneous_computing`.

parallel problems, such as graphical applications, video games, and image processing.[2] The video card technology has experienced a rapid development mainly driven by the video game industry. Table 1 reports the GigaFLOPS (i.e., billions of floating point operations per second) for two different GPUs together with two processors (CPUs) used in this study.

TABLE 1. Performance of GPU and CPU used in the paper.

| Type of video card | Stream Cores | Single-precision GFLOPS |
|---|---|---|
| NVIDIA GTS 450 | 192 | 601.34 |
| NVIDIA GT 520M | 48 | 155.51 |
| Intel Core i7-920 | 4 | 31.65 |
| Intel Core i7-2630QM | 4 | 48.24 |

In February 2012, Microsoft released C++ AMP as an extension of Visual Studio 2012 which allows to use the massively parallel and heterogeneous hardware available nowadays. Microsoft is not the only entity active in parallel computation, there are at least two other approaches: the Compute Unified Device Architecture (CUDA) of NVIDIA and the Open Computing Language (OpenCL) of the Khronos Group.

The CUDA (or, more precisely, C for CUDA) approach is an extension of C and although very fast, as documented in Aldrich et al. (2011), it suffers from vendor lock-in, since it only works with NVIDIA GPUs and cannot be used with, for instance, ATI GPUs.[3] OpenCL is designed for heterogeneous parallel computing, and since it is not tied to any specific video card manufacturer, it can work on NVIDIA and ATI GPUs. The main disadvantage thereof is that OpenCL is based on C rather than C++ and, consequently, involves significantly more manual intervention and low-level programming, see Aldrich et al. (2011) for a discussion.

C++ AMP, in contrast, simply requires a generic GPU and it abstracts the accelerators for the user. In other words, when using C++ AMP, the user does not need to worry about the specific kind of accelerator available in the system. More importantly, the programmer does not need to know or use manual memory management (different for each GPU), since it is performed automatically. This allows to write more natural and high-level programs.

As an example, let us consider a simple case of matrix multiplication $\mathbf{C} = \mathbf{AB}$, where the input matrices $\mathbf{A}$ and $\mathbf{B}$ are conformable, and $\mathbf{C}$ is the output matrix. Let us only focus on the memory management part and

[2]See `http://www.gregcons.com/CppAmp/OverviewAndCppAMPApproach.pdf`.

[3]While Thrust, a parallel algorithms library which resembles the C++ Standard Template Library (STL), improves upon the low-level approach of C for CUDA, it still suffers from the same vendor lock-in.

compare the source codes given in Listing 1 (CUDA) and Listing 2 (C++ AMP). In each listing, the views `d_A`, `d_B`, and `d_C` (available on the GPU) alias the data in **A**, **B**, and **C** (available on the CPU), respectively. While the intent and the meaning of the C++ AMP code are relatively clear, the CUDA code involves a significant amount of low-level detail, such as manual memory allocation and deallocation, manual memory access through pointers (representing values stored in the computer memory using their addresses), manual memory transfer (which also requires manually specifying the memory addresses using pointers and the direction of the transfer), and even such low-level details as the size of a single-precision floating-point data type `float` using the `sizeof` operator. This level of complexity is not only completely unnecessary in the age of modern compilers well capable of automatic type inference, but also more error-prone. [4] This is example is intended to show why a low-level approach can pose a barrier to the widespread adoption of GPU technology in the economics and econometrics community.

```
// "A" and "B" are square matrices
// "size" is the row dimension of each matrix
// allocation
cudaError_t err;
float *d_A, *d_B, *d_C;
err = cudaMalloc(&d_A, size * size * sizeof(float)); // input matrix
err = cudaMemcpy(d_A, A, size * size * sizeof(float), cudaMemcpyHostToDevice);
err = cudaMalloc(&d_B, size * size * sizeof(float)); // input matrix
err = cudaMemcpy(d_B, B, size * size * sizeof(float), cudaMemcpyHostToDevice);
err = cudaMalloc(&d_C, size * size * sizeof(float)); // output matrix
// deallocation
err = cudaMemcpy(C, d_C, size * size * sizeof(float), cudaMemcpyDeviceToHost);
err = cudaFree(d_A);
err = cudaFree(d_B);
err = cudaFree(d_C);
```

LISTING 1. CUDA sample

[4]For instance, a programmer using C for CUDA has to remember to deallocate the previously allocated memory – failure to do so may result in memory leaks; in contrast, C++ AMP follows the common C++ approach of relying on the Resource Acquisition Is Initialization (RAII) technique for automatic resource management, see Stroustrup (2000, section 14.4).

```cpp
// "A" and "B" are square matrices
// "size" is the row dimension of each matrix
array_view<const float, 2> d_A(size, size, A); // input
    matrix
array_view<const float, 2> d_B(size, size, B); // input
    matrix
array_view<float, 2> d_C(size, size, C); // output matrix
d_C.discard_data();
```

LISTING 2. C++ AMP sample

For further details, see "C++ AMP for the CUDA Programmer" by Steve Deitz,[5] which is also the source of the preceding example.

Another big advantage of C++ AMP is its flexibility, boosting developer productivity. It is not necessary to write two different implementations of the same program: one for the NVIDIA GPUs (e.g., using CUDA) and another one for the AMD GPUs (e.g., using OpenCL). It is enough to write one general implementation, since C++ AMP automatically adapts to the specific hardware available on the target machine. In the case of multiple GPUs in the system (e.g., one integrated within the CPU system and another one discrete), they can be used at the same time, even if they come from different vendors (e.g., an ATI GPU, an Intel GPU, and an NVIDIA GPU).

To date, the adoption of GPU computing technology in economics (and econometrics) has been relatively slow compared to other fields. For a literature review, see Morozov and Mathur (2011). This fact can be explained by the need to learn a new syntax, CUDA or OpenCL, the difficulty involved in implementing a different program for each different device, and the more error-prone programming involved in lower-level solutions, as demonstrated in the preceding example. We believe that C++ AMP is a solution to these problems and that it can help to promote and spread parallel programming in particular and heterogeneous programming in general in the economics and econometrics community.

To show the potential of this approach we replicate the exercise of Aldrich et al. (2011), which uses the value function iteration (VFI henceforth), an algorithm easy to express as a data parallel computational problem, to solve a dynamic equilibrium model.

We find that using VFI with binary search the (optimized) CUDA code is a little bit faster than the naive (unoptimized) implementation in C++ AMP just in one case. C++ AMP is much more powerful than CUDA in a grid search with a Howard improvement step: in this case C++ AMP is up to 5 times faster than the CUDA code.

The rest of the paper is organized as follows. Section 2 describes the basic idea of parallelization and heterogeneous programming. Section 3 presents

[5]`http://blogs.msdn.com/b/nativeconcurrency/archive/2012/04/11/c-amp-for-the-cuda-programmer.aspx`

the RBC model used in Aldrich et al. (2011). Section 4 reports some numerical results and a comparison between the two approaches. Section 5 concludes.

## 2. Parallelization and Heterogeneous Computing

In this paper we closely follow the algorithm of Aldrich et al. (2011), providing a simple parallelization scheme for GPUs (here, generalized for an arbitrary accelerator) for the VFI:

(1) Get the number of processors available, $P$, in the accelerator.
(2) Set a number of grid points, $N$, apportioning the state space such that $N = N_k \times N_z$, assigning $N_k$ points to capital and $N_z$ points to productivity.
(3) Allot $N$ grid points to each of the $P$ processors of the accelerator.
(4) Set $V^0$ to an initial guess.
(5) Copy $V^0$ to the memory of the accelerator.
(6) Compute $V^{i+1}$ , given $V^i$, using each processor for its alloted share.[6]
(7) Repeat step 6 until convergence: $\|V^{i+1} - V^i\| < \varepsilon$
(8) Copy $V^i$ from the accelerator memory to the main memory.

The practical coding is easier compared to CUDA or OpenCL.[7] For example, using CUDA and OpenCL one needs to spend quite a bit of time learning the details of manual memory management of the GPU. Since those are specific to each architecture, this can be a non-trivial problem, see Morozov and Mathur (2011). In C++ AMP memory management is automatic, and so the user has no need to learn the details of each specific architecture.

Step one of the algorithm, which requires the number of processors in the GPUs, is automatically handled in C++ AMP. The same applies for step two, the division of the $N$ grid points among the $P$ processors of the GPUs, which is a tuning parameter in CUDA (see Morozov and Mathur (2011) for a nice and thoughtful discussion), which is automatically handled with our approach.

## 3. An Application: RBC Model revisited

As our illustrative example we use the basic RBC model studied in Aldrich et al. (2011). In this model a representative household maximizes the utility function choosing consumption $\{c_t\}_{t=0}^{\infty}$ and capital $\{k_t\}_{t=0}^{\infty}$:

$$\max_{\{c_t\}_{t=0}^{\infty}} \mathbf{E}_0 \left[ \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\eta}}{1-\eta} \right], \tag{1}$$

[6]Shared memory access ensures that upon termination of this step each processor can read $V^{i+1}$.

[7]The code for our application is available from the authors upon request and will be made available under an open-source license.

where $\mathbf{E}_0$ denotes the conditional expectation operator, $\beta$ the discount factor, and $\eta$ risk aversion. The budget constraint is:

$$c_t + i_t = \omega_t + r_t k_t, \tag{2}$$

with $\omega_t$ being the wage paid for the unit of labor that the household (inelastically) supplies to the market, $r_t$ the rental rate of capital, and $i_t$ the investment. The law of motion for capital accumulation is:

$$k_{t+1} = (1-\delta)k_t + i_t, \tag{3}$$

where $\delta$ is the depreciation factor. The technology of a representative firm is $y_t = z_t k_t^\alpha$, where productivity $z_t$ follows an AR(1) process in logs:

$$\log z_t = \rho \log z_{t-1} + \varepsilon_t, \qquad \varepsilon_t \sim N(0, \sigma^2). \tag{4}$$

The resource constraint of the economy is

$$k_{t+1} + c_t = z_t k_t^\alpha + (1-\delta)k_t. \tag{5}$$

Given that the welfare theorems hold in this economy, we focus on solving the social planner's problem, which can be equivalently stated in terms of a value function $V$ and its Bellman equation

$$\begin{aligned} V(k,z) &= \max_c \left\{ \frac{c^{1-\eta}}{1-\eta} + \beta \mathbf{E}[V(k', z')|z] \right\} \\ s.t.\ k' &= zk^\alpha + (1-\delta)k - c \end{aligned} \tag{6}$$

which can be solved with VFI, which is straightforward to parallelize.

For benchmarking purposes we use the same parameter values as in Aldrich et al. (2011): $\alpha = 0.35$, $\beta = 0.984$, $\delta = 0.01$, $\eta = 2$, $\rho = 0.95$, and $\sigma = 0.005$.

## 4. Results

To implement the C++ AMP program that solves equation (6) we use Microsoft Visual Studio 2012 Beta. The CUDA code that solves the same problem, kindly provided at the following location, `http://www.ealdrich.com/Research/GPUVFI/`, is compiled using Visual Studio 2010 since the CUDA code cannot yet be compiled on Visual Studio 2012.

We use two test machines. The first is a generic PC with Intel Core i7-920 CPU and an NVIDIA GeForce GTS 450 which is an entry-level video card, with a total of 192 stream cores. The second is a standard laptop with Intel Core i7-2630QM and a GeForce 520M with 48 stream cores. Specifications of test machines, both running Windows Server 2008 R2, are reported in Table 1.

We run two experiments. The first one uses VFI with binary search, see Heer and Maussner (2005) for a discussion of this algorithm. In this experiment we increase the capital and productivity points proportionally until a grid of 1,572,864 is reached. The productivity process is discretized using the procedure of Tauchen (1986). We report the execution time of the CUDA and the C++ AMP programs in seconds, see Tables 2 and 3. In the second experiment we use the Howard improvement method and grid search; as pointed out by Aldrich et al. (2011), this algorithm yields a lower return to parallelization when using CUDA. We run their experiment with the value function maximized every $n$-th iteration of the algorithm, where $n$ is a tuning parameter decided by the user, see Tables 4 to 7.

We start with the utility of the representative household in the deterministic steady state as our $V^0$ and the convergence criterion is $\|V^{i+1} - V^i\| < (1 - \beta) \times 10^{-5}$, since we are working in single precision. We will come back to this point in Section 5.

Tables 2 and 3 report the results for the binary search algorithm with an increasing number of capital and productivity points.

TABLE 2. Observed times (in seconds) for binary value function on GeForce 450 GTS

| $N$ | *2,048* ($N_k$=512,$N_z$=4) | *8,192* ($N_k$=1,024,$N_z$=8) | *32,768* ($N_k$=2,048,$N_z$=16) | *98,304* ($N_k$=3072,$N_z$=32) |
|---|---|---|---|---|
| CUDA | 1.278 | 1.436 | 5.041 | 19.175 |
| C++ AMP | 0.831 | 1.005 | 2.228 | 14.857 |
| $N$ | *262,144* ($N_k$=4096,$N_z$=64) | *655,360* ($N_k$=5120,$N_z$=128) | *1,572,864* ($N_k$=6144,$N_z$=256) | |
| CUDA | 105.902 | 504.434 | 2305.667 | |
| C++ AMP | 102.369 | 531.524 | 2389.653 | |

TABLE 3. Observed times (in seconds) for binary value function on GeForce 520M

| $N$ | *2,048* ($N_k$=512,$N_z$=4) | *8,192* ($N_k$=1,024,$N_z$=8) | *32,768* ($N_k$=2,048,$N_z$=16) | *98,304* ($N_k$=3072,$N_z$=32) |
|---|---|---|---|---|
| CUDA | 1.888 | 2.660 | 15.811 | 71.852 |
| C++ AMP | 1.419 | 2.059 | 7.581 | 58.172 |
| $N$ | *262,144* ($N_k$=4096,$N_z$=64) | *655,360* ($N_k$=5120,$N_z$=128) | *1,572,864* ($N_k$=6144,$N_z$=256) | |
| CUDA | 369.252 | 1623.091 | N.A. | |
| C++ AMP | 336.367 | 1617.720 | 5435.050 | |

The main result of Table 2 is that the optimized CUDA code (exploiting memory access pattern, called tiling or blocking, to improve performance) is faster than the naive (unoptimized) C++ AMP code on the GeForce GTS 450 GPU, but this difference is not very large. Using the GT520M, see Table 3, characterized by less cores, the picture is different. In this case the C++ AMP program is faster than the CUDA one, moreover we experience problems in convergence for CUDA as soon as the grid reaches 1,572,864 points. This suggests that with such limited hardware C++ AMP behaves better than CUDA; this point will require further research. It is worth noting that C++ AMP can be used in two different ways. The first one is the naive implementation which is very easy and quite similar to traditional C++ code and is the programming approach used in this paper. The second one is the tiling (blocking) optimization, as in the CUDA version, which is more complex, but has potential to be significantly faster than the naive implementation.

In Tables 4 to 7 we report the results for the grid search algorithm with a Howard step. The value function is maximized only every $n$-th iteration of the algorithm. In our experiment we set $n = \{10, 20\}$, see Tables 4, 5, 6, and 7, respectively.

TABLE 4. Observed times (in seconds) for grid search algorithm, with a Howard step every 10 iterations. GPU GeForce GTS 450, CPU Core i7-920

| *N* | *32* | *64* | *128* | *256* | *512* | *1,024* |
|---|---|---|---|---|---|---|
| CUDA (GPU) | 1.207 | 1.171 | 1.077 | 1.143 | 1.161 | 1.929 |
| C++ AMP (GPU) | 0.457 | 0.518 | 0.510 | 0.652 | 0.707 | 0.763 |
| C++ (CPU) | 0.020 | 0.052 | 0.174 | 0.494 | 1.482 | 4.819 |
| *N* | *2,048* | *4,096* | *8,192* | *16,384* | *32,768* | *65,536* |
| CUDA (GPU) | 3.806 | 10.997 | 42.988 | 170.344 | 625.743 | 2,680.412 |
| C++ AMP (GPU) | 1.088 | 2.242 | 6.361 | 28.558 | 119.291 | 448.222 |
| C++ (CPU) | 18.102 | 74.782 | 270.553 | 1,147.977 | 4,662.768 | 17,896.416 |

TABLE 5. Observed times (in seconds) for grid search algorithm, with a Howard step every 10 iterations. GPU GeForce 520M, CPU Core i7-2630QM

| *N* | *32* | *64* | *128* | *256* | *512* | *1,024* |
|---|---|---|---|---|---|---|
| CUDA (GPU) | 1.531 | 1.539 | 1.604 | 1.755 | 2.253 | 3.685 |
| C++ AMP (GPU) | 1.232 | 1.217 | 1.234 | 1.326 | 1.435 | 1.716 |
| C++ (CPU) | 0.016 | 0.031 | 0.109 | 0.374 | 1.326 | 4.165 |
| *N* | *2,048* | *4,096* | *8,192* | *16,384* | *32,768* | *65,536* |
| CUDA (GPU) | 12.622 | 45.758 | 175.330 | 719.923 | 3,087.640 | 11,856.700 |
| C++ AMP (GPU) | 3.369 | 10.233 | 34.585 | 151.851 | 580.648 | 2,393.450 |
| C++ (CPU) | 16.162 | 67.907 | 239.429 | 1,121.230 | 3,920.550 | 16,454.090 |

The main finding is that the naive implementation using C++ AMP is very fast, up to 5 times faster than the CUDA implementation. As explained in Aldrich et al. (2011) the CPU approach is much slower than the GPU one.

TABLE 6. Observed times (in seconds) for grid search algorithm, with a Howard step every 20 iterations. GPU GeForce GTS 450, CPU Core i7-920

| *N* | *32* | *64* | *128* | *256* | *512* | *1,024* |
|---|---|---|---|---|---|---|
| CUDA (GPU) | 1.136 | 1.007 | 1.019 | 1.171 | 1.204 | 1.442 |
| C++ AMP (GPU) | 0.489 | 0.496 | 0.505 | 0.598 | 0.633 | 0.646 |
| C++ (CPU) | 0.023 | 0.056 | 0.143 | 0.382 | 0.997 | 3.252 |
| *N* | *2,048* | *4,096* | *8,192* | *16,384* | *32,768* | *65,536* |
| CUDA (GPU) | 2.492 | 6.615 | 22.617 | 88.949 | 353.202 | 1,452.964 |
| C++ AMP (GPU) | 0.819 | 1.369 | 4.192 | 17.561 | 53.414 | 266.218 |
| C++ (CPU) | 12.372 | 48.076 | 148.888 | 638.175 | 2,542.604 | 10,170.048 |

TABLE 7. Observed times (in seconds) for grid search algorithm, Howard step every 20 iterations. GPU GeForce 520M, CPU Core i7-2630QM

| *N* | *32* | *64* | *128* | *256* | *512* | *1,024* |
|---|---|---|---|---|---|---|
| CUDA (GPU) | 1.512 | 1.541 | 1.537 | 1.634 | 1.932 | 3.082 |
| C++ AMP (GPU) | 1.236 | 1.232 | 1.248 | 1.341 | 1.388 | 1.575 |
| C++ (CPU) | 0.016 | 0.031 | 0.062 | 0.265 | 0.905 | 2.933 |
| *N* | *2,048* | *4,096* | *8,192* | *16,384* | *32,768* | *65,536* |
| CUDA (GPU) | 7.486 | 24.242 | 93.862 | 373.019 | 1485.410 | 6126.650 |
| C++ AMP (GPU) | 2.480 | 5.694 | 18.049 | 81.010 | 322.327 | 1286.190 |
| C++ (CPU) | 10.920 | 44.070 | 135.439 | 580.181 | 2222.960 | 9754.130 |

## 5. CONCLUSION

We propose and present a new approach for massively parallel programming in economics, C++ AMP. We show that this approach has a lot of potential. First, the programming simplicity, second the hardware generality (C++ AMP adapts to different GPUs automatically) and third the performance (in some cases it is much faster than a specialized approach such as CUDA).

As a future research project we are also interested in investigating the reasons behind the 5x speed-up in select cases. A potential solution would involve comparing the generated PTX (Parallel Thread Execution) code on the CUDA side with the HLSL (High Level Shader Language) code on the C++ AMP side. However, the current (beta) version of C++ AMP does not offer the capability to view the generated HLSL code.

Our results are solely intended as an illustration of a lower bound of this technology for two reasons. First, it is still in the early beta testing phase and a lot of improvements are expected with the final release by the end of 2012. Second, since C++ AMP is an open standard, it can be implemented by different vendors, such as ATI, for different platforms, such as Unix-like systems.

At the moment there is still a drawback to using double precision computation. For the GPUs used in this study, C++ AMP offers full support for double precision computation on Windows NT 6.2 (i.e., "Windows 8" and "Windows Server 2012") only. The upcoming release of new drivers from the major vendors, such as NVIDIA and ATI, should solve this problem for the current Windows operating systems (NT 6.1 – i.e., "Windows 7" and "Windows Server 2008 R2").[8] For the other high-end GPUs, such as GeForce GTX 460, full double precision support is already available.

Since heterogeneous computing is a fast-evolving field we expect that additional findings will be forthcoming soon.

## References

Aldrich, E. M., Fernández-Villaverde, J., Gallant, A. R., and Rubio Ramırez, J. F. (2011). Tapping the Supercomputer Under Your Desk: Solving Dynamic Equilibrium Models with Graphics Processors. *Journal of Economic Dynamics and Control*, Vol. 35:pp. 386–393.

Heer, B. and Maussner, A. (2005). *Dynamic General Equilibrium Modelling: Computational Methods and Applications*. Springer, Berlin.

Morozov, S. and Mathur, S. (2011). Massively Parallel Computation Using Graphics Processors with Application to Optimal Experimentation in Dynamic Control. *Computational Economics*, pages 1–32.

Stroustrup, B. (2000). *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition.

Tauchen, G. (1986). Finite State Markov-Chain Approximations to Univariate and Vector Autoregressions. *Economics Letters*, Vol. 20:pp. 177 – 181.

[8]For more details, see `http://blogs.msdn.com/b/nativeconcurrency/archive/2012/02/07/double-precision-support-in-c-amp.aspx`.