Computational Economics manuscript No.
(will be inserted by the editor)

Heterogeneous Computing In Economics: A Simplified Approach

Matt P. Dziubinski · Stefano Grassi

December 6, 2013

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.

Keywords CUDA · C++ · C++ AMP · DSGE models · Econometrics · Heterogeneous 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 acknowledge financial support by the Center for Research in Econometric Analysis of Time Series, CREATES, funded by the Danish National Research Foundation.

Matt P. Dziubinski
Department of Mathematical Sciences, Aalborg University and CREATES, Denmark
E-mail: matt@math.aau.dk

Stefano Grassi
CREATES, Department of Economics and Business, Aarhus University, Fuglesangs Allé 4, 8210 Aarhus V, Denmark.
Tel.: +45 8716 5905.
E-mail: sgrassi@creates.au.dk

[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`.

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 parallel problems, where we assign an individual data element to a separate logical core for processing, applications include graphics, video games, and image processing.[2] The video card technology has experienced a rapid development mainly driven by 3D rendering (with notable applications in the video game industry), see Boyd (2008). Table 1 reports the GigaFLOPS (i.e., billions of floating point operations per second) for the GPU and the CPU used in this study.

Table 1: Performance of the GPU and the CPU used in this study.

Type of video card	Stream Cores	Single-precision GFLOPS
NVIDIA GTS 450	192	601.34
Intel Core i7-920	4	31.65

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.

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++

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

[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.

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 simultaneously, 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, see Morozov and Mathur (2012) for a literature review. This is somewhat surprising, given the fact that there is a large number of economic and econometric applications where parallel computing can be applied, see Creel and Goffe (2008). The low diffusion of this technology in the economics and econometrics literature, according to Creel (2005), is related to the steep learning curve of a dedicated programming language and expensive hardware. Modern GPUs can easily solve the second problem (hardware costs are relatively low), but the the first issue still remains open.

We think that C++ AMP is a solution to this problem 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 program is slower than the naive (unoptimized) implementation using C++ AMP. C++ AMP is also faster in a grid search with a Howard improvement step. In both cases the C++ AMP program is almost 3 times faster than the CUDA one.

The rest of the paper is organized as follows. Section 2 provides a simple example that shows the simplicity of the proposed approach compared to CUDA. Section 3 describes the basic idea of parallelization and heterogeneous programming applied to VFI. Section 4 presents the RBC model used in Aldrich et al. (2011). Section 5 reports the time comparison between the two approaches. Section 6 concludes. A complete example of the C++ AMP programming style is presented in the Appendix.

2 A Simple Example

This example is intended to show how a low-level programming approach, such as CUDA, can pose a barrier to the widespread adoption of GPU technology in the economics and econometrics community, and how a high-level C++ AMP approach solves the problem.

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

```
// "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. Note that error handling code has been omitted for conciseness.

```
// "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.

3 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:

[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).

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

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 (2012). 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 (2012) for a nice and thoughtful discussion), which is automatically handled with our approach.

4 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 $\{k_t\}_{t=0}^{\infty}$:

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

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

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

with ω_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:

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

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

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

where δ 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:

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

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

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

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 $\sigma = 0.005$.

5 Results

To implement the C++ AMP program that solves equation (6) we use Microsoft Visual Studio 2012 RTM. 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 RTM.

Out test machine is a generic PC with Intel Core i7-920 CPU and NVIDIA GeForce GTS 450 which is an entry-level video card, with a total of 192 stream cores. The test machine runs Windows Server 2012, for specifications see also Table 1.

We run two experiments. The first one uses VFI with binary search, for an introduction to this algorithm see Heer and Maussner (2005) . In this experiment the number of capital points, N_k, is increased proportionally until a grid of 262,144 points is reached. The productivity process is discretized using the procedure of Tauchen (1986). We report the memory allocation and solution time of the CUDA and the C++ AMP programs in seconds, see Table 2. The Table also reports the total execution time for the CPU.

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. In this case the number of capital points, N_k, is also increased proportionally until a grid of 262,144 points is reached. 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 3 and 4.

We start with the utility of the representative household in the deterministic steady state as our V^0 and the convergence criterion is $\|V^{i+1} - V^{i}\| < (1-\beta) \times 10^{-8}$, since we are working in double precision. We will come back to this point in Section 6.

Table 2 reports the results for the binary search algorithm with an increasing number of capital points.

Table 2: Observed times (in seconds) for Binary Search. Also reported are the memory allocation (Mem-Alloc) time and the solution time (Solution) for the CUDA and the C++ AMP programs. For CPU only the total execution time is reported. GPU GeForce GTS 450, CPU Core i7-920.

N_k		*32*	*64*	*128*	*256*	*512*	*1,024*	*2,048*	*4,096*	*8,192*	*16,384*	*32,768*	*65,536*
CUDA (GPU)	Mem-Alloc	1.62	1.62	1.62	1.62	1.62	1.62	1.62	1.62	1.62	1.62	1.62	1.62
CUDA (GPU)	Solution	0.19	0.28	0.25	0.37	0.62	1.22	2.80	5.10	10.78	22.82	48.67	103.42
C++ AMP (GPU)	Mem-Alloc	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06
C++ AMP (GPU)	Solution	1.93	1.72	1.72	1.89	1.96	2.34	2.72	3.67	5.86	10.44	20.39	41.81
C++ (CPU)	Total	0.10	0.24	0.51	1.16	3.04	8.00	23.31	80.32	288.71	1,477.25	5,692.20	13,930.8

The main result of Table 2 is that the optimized CUDA program (exploiting memory access pattern, called tiling or blocking, to improve performance) is slower than the naive (unoptimized) C++ AMP program on the GeForce GTS 450 GPU.[8] Two more results have to be emphasized. First, the memory allocation step is much faster in the C++ AMP program than in the CUDA one. Second, for small grid the CUDA program is faster than the C++ AMP one. This advantage is lost as soon as the grid increases and in the end the C++ AMP program is almost 3 times faster than the CUDA one. Hence, we conclude that C++ AMP might offer better scalability for larger computational problems.

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 the potential to be significantly faster than the naive implementation for specific problems, see `http://msdn.microsoft.com/en-us/magazine/hh882447.aspx`

In Tables 3 and 4 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 3 and 4, respectively.

[8] In a previous version of the paper we also find that the single-precision CUDA program, in this particular case, is slightly faster than the single-precision C++ AMP program.

Table 3: Observed times (in seconds) for grid search algorithm, with a Howard step every 10 iterations. Also reported are the memory allocation (Mem-Alloc) time and the solution time (Solution) for the CUDA and the C++ AMP programs. For CPU only the total execution time is reported. GPU GeForce GTS 450, CPU Core i7-920.

N_k		*32*	*64*	*128*	*256*	*512*	*1,024*
CUDA (GPU)	Mem-Alloc	1.62	1.62	1.62	1.62	1.62	1.62
CUDA (GPU)	Solution	0.16	0.17	1.90	0.37	1.11	3.80
C++ AMP (GPU)	Mem-Alloc	0.06	0.06	0.06	0.06	0.06	0.06
C++ AMP (GPU)	Solution	1.81	1.92	1.67	1.79	2.42	4.11
C++ (CPU)	Total	0.04	0.09	0.22	0.61	1.70	7.48
N_k		*2,048*	*4,096*	*8,192*	*16,384*	*32,768*	*65,536*
CUDA (GPU)	Mem-Alloc	1.62	1.62	1.62	1.62	1.62	1.62
CUDA (GPU)	Solution	14.32	55.79	220.38	879.56	3,503.06	13,593.30
C++ AMP (GPU)	Mem-Alloc	0.06	0.06	0.06	0.06	0.06	0.06
C++ AMP (GPU)	Solution	8.93	28.52	109.17	365.93	1,438.98	5,702.69
C++ (CPU)	Total	24.87	91.42	440.68	1,743.93	7,137.80	27,239.2

Table 4: Observed times (in seconds) for grid search algorithm, with a Howard step every 20 iterations. Also reported are the memory allocation (Mem-Alloc) time and the solution time (Solution) for the CUDA and the C++ AMP programs. For CPU only the total execution time is reported. GPU GeForce GTS 450, CPU Core i7-920.

N_k		*32*	*64*	*128*	*256*	*512*	*1,024*
CUDA (GPU)	Mem-Alloc	1.62	1.62	1.62	1.62	1.62	1.62
CUDA (GPU)	Solution	0.12	0.13	0.15	0.26	0.71	2.07
C++ AMP (GPU)	Mem-Alloc	0.06	0.06	0.06	0.06	0.06	0.06
C++ AMP (GPU)	Solution	1.88	1.83	1.69	1.752	2.21	3.10
C++ (CPU)	Total	0.03	0.07	0.16	0.45	1.25	3.91
N_k		*2,048*	*4,096*	*8,192*	*16,384*	*32,768*	*65,536*
CUDA (GPU)	Mem-Alloc	1.62	1.62	1.62	1.62	1.62	1.62
CUDA (GPU)	Solution	7.57	28.75	112.89	446.97	1,787.62	7,108.05
C++ AMP (GPU)	Mem-Alloc	0.06	0.06	0.06	0.06	0.06	0.06
C++ AMP (GPU)	Solution	5.62	14.64	49.59	187.93	735.45	2,907.54
C++ (CPU)	Total	14.67	52.74	216.98	930.62	3,934.37	14,913.3

As in the binary search case the naive implementation of C++ AMP is very fast, almost 3 times faster than the CUDA implementation.[9] As explained in Aldrich et al. (2011) the CPU approach is much slower than the GPU one.

6 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 the considered cases it is much faster than a specialized approach such as CUDA).

[9] In a previous version of the paper, working in single precision, we also find that the C++ AMP program in this case is more than 5 times faster than the CUDA one.

As a future research project we are also interested in investigating the reasons behind the speed-up observed for the C++ AMP program 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 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 a relative new technology and 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").[10] For the other high-end GPUs, such as GeForce GTX 460, full double precision support is already available. To test the reliability of the C++ AMP program's results in all of the experiments, we carried out a comparison between the CUDA, the C++ AMP, and the CPU output, finding no differences.

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

Appendix. The C++ AMP Programming Style

To show the similarities and differences between the standard C++, ISO (2011) and C++ AMP we provide as an example the code to obtain a vector containing the element-by-element squares of the numbers in a given input vector (a sum of the elements of the output vector would give the sum of squares, often useful in econometrics applications). Let's take vector $Vector = \{1, 2, 3, 4, 5\}$ and calculate the corresponding squares vector $Squares = \{1, 4, 9, 16, 25\}$. Listing 3 reports the C++ and the C++ AMP code. We define three vectors: **hostVector** is the input vector, while **hostSquares_usingCPU** and **hostSquares_usingGPU** are the output vectors (both stored in the host's memory), obtained using the host (CPU) and the device (GPU), respectively. Functions **hostSquares** and **deviceSquares** perform squaring on the host and the device, respectively. Finally the results are compared.

Note that from the point of view of the client code (here: function **main**), the interface (and, consequently, the use) of **hostSquares** is identical to that of **deviceSquares**. This illustrates the fact that C++ AMP can be used to incrementally introduce parallelism to an existing C++ code base – whenever necessary and without breaking changes.

The only differences are in the internal implementation details of both functions – while **hostSquares** uses standard C++ constructs, **deviceSquares** uses parallel con-

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

structs from the `concurrency` namespace (made available via the inclusion of the *amp.h* header).

Note that data-parallel applications are an especially good fit for GPU computing. In terms of Flynn's taxonomy, Flynn (1972), this can be related to Single Instruction, Multiple Data (SIMD) – or, more precisely, a more general category thereof – SPMD (single program, multiple data). In terms of our example, the single program (here: multiply the values in the input vector by themselves, storing thus obtained squared values in the output vector) is applied to multiple data (elements of the input vector), where the lack of data dependence (between the distinct elements of the input vector) allows to spread the work onto multiple threads.

```cpp
#include <iostream>
#include <vector>
#include <cassert>
#include <amp.h>
typedef std::vector<int> VectorType;
// squares using host (CPU)
void hostSquares(const VectorType & hostVector, VectorType & hostSquares)
{
  int size = hostVector.size();
  for (int i = 0; i < size; ++i)
  {
    hostSquares[i] = hostVector[i] * hostVector[i];
  }
}
// squares using accelerator (GPU)
void deviceSquares(const VectorType & hostVector, VectorType & hostSquares)
{
  int size = hostVector.size();
  concurrency::array_view<const int, 1> deviceVector(size, hostVector);
  concurrency::array_view<int, 1> deviceSquares(size,hostSquares);
  deviceSquares.discard_data();
  concurrency::parallel_for_each(deviceSquares.extent, [=](concurrency::index<1> i) restrict(amp)
  {
    deviceSquares[i] = deviceVector[i] * deviceVector[i];
  });
}
int main()
{
  // fill input vector with data
  VectorType hostVector { 1, 2, 3, 4, 5 };
  // obtain output squares vector using CPU
  VectorType hostSquares_usingCPU(hostVector.size());
  hostSquares(hostVector, hostSquares_usingCPU);
  // obtain output squares vector using GPU
  VectorType hostSquares_usingGPU(hostVector.size());
  deviceSquares(hostVector, hostSquares_usingGPU);
  // check if the results are identical
  assert(hostSquares_usingCPU == hostSquares_usingGPU);
}
```

Listing 3: C++ and C++ AMP source code

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. 35:386 – 393.

Boyd, C. (2008). Data-parallel computing. *Queue*, Vol. 6:30–39.

Creel, M. (2005). User-Friendly Parallel Computations with Econometric Examples. *Computational Economics*, Vol. 26:107 – 128.

Creel, M. and Goffe, W. L. (2008). Multi-core CPUs, Clusters, and Grid Computing: A Tutorial. *Computational Economics*, Vol. 32:353 – 382.

Flynn, M. (1972). Some computer organizations and their effectiveness. *IEEE Transaction Computing*, Vol. 21:948 – 960.

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

ISO (2011). *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization.

Morozov, S. and Mathur, S. (2012). Massively Parallel Computation Using Graphics Processors with Application to Optimal Experimentation in Dynamic Control. *Computational Economics*, Vol. 21:151 – 182.

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. 20:177 – 181.