SIGFIRM Working Paper No. 15

Massively Parallel Computing in Economics

Eric M. Aldrich, University of California, Santa Cruz

January 2013

SIGFIRM-UCSC
Engineering-2, Room 403E
1156 High Street
Santa Cruz, CA 95064
831-459-2523
Sigfirm.ucsc.edu

The Sury Initiative for Global Finance and International Risk Management (SIGFIRM) at the University of California, Santa Cruz, addresses new challenges in global financial markets. Guiding practitioners and policymakers in a world of increased uncertainty, globalization and use of information technology, SIGFIRM supports research that offers new insights into the tradeoffs between risk and rewards through innovative techniques and analytical methods. Specific areas of interest include risk management models, models of human behavior in financial markets (the underpinnings of “behavioral finance”), integrated analyses of global systemic risks, and analysis of financial markets using Internet data mining.
Massively Parallel Computing in Economics

Eric M. Aldrich*

Department of Economics
University of California, Santa Cruz

January 2, 2013

Abstract

This paper discusses issues related to parallel computing in Economics. It highlights new methodologies and resources that are available for solving and estimating economic models and emphasizes situations when they are useful and others where they are impractical. Two examples illustrate the different ways parallel methods can be employed to speed computation as well as their limitations.

Keywords: Parallel Computing, GPU Computing, Dynamic Programming, Generalized Stochastic Simulation Algorithm.

JEL Classification: A33, C60, C63, C68, C88.

*Email: ealdrich@ucsc.com
1 Introduction

Recently developed computer hardware and software has resulted in a new revolution in scientific computing in the last decade. As microprocessors became increasingly limited in terms of speed gains in the early 2000s, the computing industry moved toward developing multi-core and multi-processor Central Processing Unit (CPU) systems. Somewhat independently, the market for high-end graphics in video games led to the development of many-core Graphical Processing Units (GPUs) in the late 1990s. These graphics cards were designed to have many individual processing units with a limited instruction set and limited memory access. The result was a set of devices with high arithmetic intensity - number of operations per byte of memory transferred.

Computational scientists were quick to recognize the value of parallel computing and to harness its power. Some time after the turn of the millennium, a number of researchers began using GPUs as parallel hardware devices for solving scientific problems. Early examples spanned the fields of molecular dynamics, astrophysics, aerospace engineering, climate studies, and mathematics to name a few. In each case scientists recognized similarities between their algorithms and the work of rendering millions of graphical pixels in parallel. In response to the uptake of GPU computing in broad scientific fields, NVIDIA released a set of software development tools in 2006, known as Compute Unified Device Architecture (CUDA). The intention of CUDA was to facilitate higher-level interaction with graphics cards and to make their resources accessible through industry standard languages, such as C and C++. This facilitated a new discipline of General Purpose GPU (GPGPU) computing, with a number of subsequent tools that have been developed and released by a variety of hardware and software vendors.

The uptake of GPGPU computing in Economics has been slow, despite the need for computational power in many economic problems. Recent examples include Aldrich (2011), Aldrich et al. (2011), Creal (2012), Creel and Kristensen (2011), Durham and Geweke (2011), Durham and Geweke (2012), Dziubinski and Grassi (2012), Fulop and Li (2012) and Lee et al. (2010). The objective of this paper will be to demonstrate the applicability of massively parallel computing to economic problems and to highlight situations in which it is most
beneficial and also of little use.

The benefits and limitations of GPGPU computing in economics will be demonstrated via two specific examples, which with very different structures. The first, a basic dynamic programming problem solved with value function iteration provides a simple framework to demonstrate the “embarrassingly” parallel nature of many problems and how their computational structure can be quickly adapted to a massively parallel framework. In this case, the speed gains are tremendous. The second example, a multi-country real business-cycle (RBC) model solved with the Generalized Stochastic Simulation Algorithm (GSSA) of Judd et al. (2011a) demonstrates more modest speed gains due to dependencies among state variables.

The landscape of scientific parallel computing is quickly changing, which is primarily attributed to its usefulness and popularity. As a result, the specific hardware and software tools used in this paper will become outdated within a short period of time. While the illustration of the concepts herein will rely to some extent on specific software and hardware platforms, the primary objective is to convey a way think about economic problems that transcends those ephemeral details. Although the future of massively parallel computing will change at a rapid pace, the way in which we adapt our algorithms to parallel devices will be much more stable. Hence, this paper will provide enough discussion of software and hardware that a researcher can become familiar with the current state of the art, but the primary focus will be on algorithmic structure and adaptation.

The structure of this paper will be as follows. Section 2 will introduce the basic concepts of GPGPU computing along with simple examples. Sections 3 and 4 will consider the dynamic programming and multi-country RBC examples mentioned above, demonstrate how the solutions can be parallelized, and report timing results. Section 5 will discuss recent developments in parallel computing and will offer a glimpse of the future of the discipline and potential changes for economic computing. Section 6 will conclude.

2 Basics of GPGPU Computing

While the objective of this paper is to discuss concepts of parallel computing that transcend the ephemeral nature of hardware and software, some discussion of current computing envi-
ronments is necessary for framing later discussion. This section will introduce the basics of massively parallel computing through a very simple example, and will provide demonstration code that can be used as a template for GPGPU research projects.

2.1 Architecture

Understanding the basics of GPU architecture facilitates design of massively parallel software. For illustrative purposes, this section will often reference the specifications of an NVIDIA Tesla C2075 GPU, a current high-end GPU intended for scientific computing.

2.1.1 Processing Hardware

GPUs are comprised of dozens to hundreds of individual processing cores. These cores, known as thread processors, are typically grouped together into several distinct multiprocessors. For example, the Tesla C2075 has a total of 448 thread processors, aggregated into groups of 32 cores per multiprocessor, yielding a total of 14 multiprocessors. Relative to CPU cores, GPU cores typically:

- Have a lower clock speed. Each Tesla C2075 core clocks in at 1.15 GHz, which is roughly 40% of current CPU clock speeds.

- Dedicate more transistors to arithmetic operations and fewer to control flow and data caching.

- Have access to less memory. A Tesla C2075 has 6 gigabytes of global memory, shared among all cores.

Clearly, where GPU processors are lacking in clock speed and memory access, they compensate with sheer quantity of compute cores. For this reason, they are ideal for computational work that has a high arithmetic intensity: many arithmetic operations for each byte of memory transfer/access. Figure 1 depicts a schematic diagram of CPU and GPU architectures, taken from Section 1.1 of NVIDIA (2012).
The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation—exactly what graphics rendering is about—and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 1-2.

Figure 1: Schematic diagram of CPU and GPU processors, taken from Section 1.1 of NVIDIA (2012).

2.1.2 Algorithmic Design

Kernels and threads are the fundamental elements of GPU computing problems. Kernels are special functions that comprise a sequence of instructions that are issued in parallel over a user specified data structure (i.e. performing a routine on each element of a vector). Each data element and corresponding kernel comprise a thread, which is an independent problem that is assigned to one GPU core.

Just as GPU cores are grouped together as multiprocessors, threads are grouped together in user-defined groups known as blocks. Thread blocks execute on exactly one multiprocessor, and typically many thread blocks are simultaneously assigned to the same multiprocessor. A diagram of this architecture is depicted in Figure 2, taken from section 1.1 of NVIDIA (2012). The scheduler on the multiprocessor then divides the user defined blocks into smaller groups of threads that correspond to the number of cores on the multiprocessor. These smaller groups of threads are known as warps - as described in NVIDIA (2012), “The term warp originates from weaving, the first parallel thread technology”. As mentioned above, each core of the multiprocessor then operates on a single thread in a warp, issuing each of the kernel instructions in parallel. This architecture is known as Single-Instruction, Multiple-Thread (SIMT).

Because GPUs employ SIMT architecture, it is important to avoid branch divergence among threads. While individual cores operate on individual threads, the parallel structure
This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available processor cores, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of processor cores as illustrated by Figure 1-4, and only the runtime system needs to know the physical processor count.

This scalable programming model allows the CUDA architecture to span a wide market range by simply scaling the number of processors and memory partitions: from the high-performance enthusiast GeForce GPUs and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs (see Appendix A for a list of all CUDA-enabled GPUs).

A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

Figure 2: Schematic diagram of thread blocks and GPU multiprocessors, taken from Section 1.1 of NVIDIA (2012).

achieves greatest efficiency when all cores execute the same instruction at the same time. Branching within threads is allowed, but asynchronicity may result in sequential execution over data elements of the warp. Given the specifications of GPU cores, sequentially execution would be horribly inefficient relative to simply performing sequential execution on the CPU.

2.1.3 Scaling

One of the wonders of GPU computing, relative to other forms of parallel CPU computing, is that it automatically scales across different hardware devices. With MPI and OpenMp, the prevailing platforms for CPU parallelism, it is necessary for users to be aware of the exact number of processors available, and (in the case of MPI) to write instructions to govern their interactions. When moving software among differing systems, it is then crucial to alter code so as to accommodate changes in hardware. GPU interfaces (discussed below),
on the other hand, allow the software designer to be agnostic about the exact architecture of the GPU - the user does nothing more than designate the size of thread blocks, which are then allocated to multiprocessors by the GPU scheduler. Although different block sizes are optimal for different GPUs (based on number of processing cores), it is not requisite to change block sizes when moving code from one GPU to another. The upshot is that the scheduler deals with scalability so that issues related to processor count and interaction on a specific device are transparent to the user. This increases the portability of massively parallel GPU software.

2.1.4 Memory

There is a distinction between CPU memory and GPU memory, the former being referred to as ‘host’ memory and the latter as ‘device’ memory. GPU kernels can only operate on data objects that are located in device memory - attempting to pass a variable in host memory as an argument to a kernel would generate an error. Thus, GPU software design often necessitates the transfer of data objects between host and device memory.

Currently, memory transfers between host and device occur over a PCIe×16 interface, which for an NVIDIA Tesla C2075 GPU translates into a data transfer bandwidth of roughly 6 megabytes per second. This is approximately 1/4th the bandwidth between common configurations of host memory and CPU at the present date. For this reason it is crucial to keep track of host-device memory transfers, since there are many situations in which they can be a speed-limiting factor for computation.

The architecture of GPU memory itself is also important. While all GPU cores share a bank of global memory, portions of the global memory are partitioned for shared use among cores on a multiprocessor. Access to this shared memory is much faster than global memory. While these issues can be beneficial to the design of parallel algorithms, the intricacies of GPU memory architecture are beyond the scope of this paper.
2.2 Software

GPU software is changing at a fast rate, which means that the details in this section will become quickly outdated. For this reason, only brief mention is made of current software. In addition, while this section is only intended to be a quick overview of available software, the next section will provide detailed examples of how to use the software within the context of a parallel computing problem.

NVIDIA was the original leader in developing a set of software tools allowing scientists to access GPUs. The CUDA C language is simply a set of functions that can be called within basic C/C++ code that allow users to interact with GPU memory and processors. CUDA C is currently the most efficient and best documented way to design massively parallel software – it is truly the state of the art. Downsides to CUDA C are that it requires low-level comfort with software design (similar to C/C++) and that it only runs on NVIDIA GPUs running the CUDA platform. The CUDA platform itself is free, but requires NVIDIA hardware. While originally designed only for C/C++, it is now possible to write CUDA C kernels for Fortran, Python and Java.

OpenCL is an open source initiative lead by Apple and promoted by the Khronos Group. The syntax of OpenCL is very similar to CUDA C, but it has the advantage of not being hardware dependent. In fact, not only can OpenCL run on a variety of GPUs (including NVIDIA GPUs), it is intended to exploit the heterogeneous processing resources of differing GPUs and CPUs simultaneously within one system. The downside to OpenCL is that it is poorly documented and has much less community support than CUDA C. In contrast to NVIDIA CUDA, it is currently very difficult to find a cohesive set of documentation that assists an average user in making an OpenCL capable system and in beginning the process or software design with OpenCL.

Beyond these two foundational GPU software tools, more and more third-party vendors are developing new tools, or adding GPU functionality within current software. Examples include the Parallel Computing Toolbox in Matlab and the CUDALink and OpenCLLink interfaces in Mathematica. New vendors, such as AccelerEyes are developing libraries that allow higher-level interaction with the GPU: their Jacket product is supposed to be a superior
parallel computing library for Matlab, and their ArrayFire product is a matrix library that allows similar high-level interaction within C, C++ and Fortran code. ArrayFire works with both the CUDA and OpenCL platforms (i.e. any GPU) and the basic version is free. For a licensing fee, users can also gain access to linear algebra and sparse grid library functions.

Similar to ArrayFire, matrix libraries such as Thrust, ViennaCL and C++Amp have been developed to allow higher-level GPU support within the context of the C and C++ languages. All are free, although each has specific limitations: e.g. Thrust only works on the CUDA platform, and, at present, C++Amp only works on the Windows operating system via Visual Studio 2012 (and hence is not free if VS2012 cannot be obtained though an academic license). While tied to NVIDIA hardware, Thrust is a well documented and well supported library which will be featured below.

2.3 Simple Example

We now turn to a simple problem that can be computed with a GPU and illustrate how it can be implemented in several computing languages. Consider the second-order polynomial

\[ y = ax^2 + bx + c. \]  

Suppose that we wish to optimize the polynomial for a finite set of values of the second-order coefficient in a specific range: \( a \in [-0.9, -0.1] \). Figure 3 depicts this range of polynomials when \( b = 2.3 \) and \( c = 5.4 \), and where the darkest line corresponds to the case \( a = -0.1 \). In this example it is trivial to determine the location of the optimum,

\[ x = \frac{b}{2a}, \]  

however to illustrate the mechanics of parallel computing we will compute the solution numerically with Newton’s Method for each \( a \in [-0.9, -0.1] \).

The remainder of this section will show how to solve this problem with Matlab, C++, CUDA, C and Thrust. All of the code can be obtained from http://www.parallelecon.com/basically-gpu/. The Matlab and C++ codes are provided merely as building blocks – they are not parallel implementations of the problem. In particular, the Matlab code serves as a baseline and demonstrates how to quickly solve the problem in a language that is familiar to most
economists. The C++ code then demonstrates how easily the solution can be translated from Matlab to C++; indeed, most economists will be surprised at the similarity of the two languages. Understanding the serial C++ implementation is crucial for CUDA C and Thrust, since these latter implementations are simply libraries that extend the C++ framework.

### 2.3.1 Matlab

Listings 1 displays the file `main.m`, which solves the optimization problem above for various values of the second-order coefficient. The block of code on lines 2-4

```matlab
nParam = 100000000;
paramMin = -0.9;
paramMax = -0.1;
paramGrid = paramMin:((paramMax-paramMin)/(nParam-1)):paramMax;
```

constructs a grid, `paramGrid`, of `nParam = 100000000` values between -0.9 and -0.1. Line 8 then allocates a vector for storing the arg max values of the polynomial at each \( a \),

```matlab
argMaxVals = zeros(nParam,1);
```

and lines 9-11 loop over each value of `paramGrid` and maximize the polynomial by calling the function `maxPoly`,

```matlab
for i = 1:nParam
    argMaxVals(i) = maxPoly(2.2, paramGrid(i), 0.00001);
end
```
argMaxVals(i) = maxPoly(2.2, paramGrid(i), 0.00001);
end

To numerically solve for the maximum at line 10, Matlab provides built-in optimization functions such as fmincon; alternatively, superior third-party software, such as KNITRO (http://www.ziena.com/knitro.htm), could be used. To keep the Matlab software similar to the implementations below, we make use of a self-written Newton solver wrapped in the function maxPoly, which is shown in Listing 2. The first line of the listing

function argMax = maxPoly(x0, coef, tol)

shows that maxPoly accepts three arguments: an initial value for x, x0, a value of the second-order coefficient, coef, and a convergence tolerance, tol. On exit, the function returns a
single value, \texttt{argMax}, which is the arg max of the function. Lines 4 and 5

\begin{verbatim}
    x = x0;
    diff = tol+1;
\end{verbatim}

initialize the arg max, \(x\), and create a variable, \texttt{diff}, which tracks the difference between
Newton iterates of \(x\). The main Newton step then occurs within the while loop between
lines 6 and 21. In particular, lines 9 and 12 compute the first and second derivatives of the
polynomial,

\begin{verbatim}
    firstDeriv = 2*coef*x + 2.3;
    secondDeriv = 2*coef;
\end{verbatim}

and line 15

\begin{verbatim}
    xNew = x - firstDeriv/secondDeriv;
\end{verbatim}

uses the derivatives to update the value of the arg max, \texttt{xNew}. Each iteration terminates by
computing the difference between the new and current iterates

\begin{verbatim}
    diff = abs(xNew - x);
\end{verbatim}

and then setting the new value of the arg max to be the current value

\begin{verbatim}
    x = xNew;
\end{verbatim}

When convergence is achieved (\texttt{diff < tol}), the function exits and returns the most recent
value of \(x\). As is seen above, the basic nature of the problem makes it very easy to solve
with few lines of code.

\subsection*{2.3.2 C++}

Listings 3 and 4 display C++ code for the polynomial optimization problem. This code
makes no direct advances towards parallelization, but sets the framework for subsequent
parallel implementations (CUDA C and Thrust) which build on C++. While most economists
are not comfortable with C++, many will be surprised by the similarity between the Matlab
and C++ code, especially the functions \texttt{maxPoly.m} and \texttt{maxPoly.cpp}.
Listing 3: C++ code for polynomial maximization problem: main.cpp

```cpp
#include <Eigen/Dense>

using namespace Eigen;

double maxPoly(double x0, double coef, double tol);

int main()
{
    // Grid for order 2 coefficient
    int nParam = 10000000;
    double paramMin = -0.9;
    double paramMax = -0.1;
    VectorXd paramGrid = VectorXd::LinSpaced(nParam, paramMin, paramMax);

    // Maximize for each coefficient
    VectorXd argMaxVals = VectorXd::Zero(nParam);
    for(int i = 0; i < nParam; ++i)
    {
        argMaxVals(i) = maxPoly(2.2, paramGrid(i), 0.00001);
    }

    return 0;
}
```
# Listing 4: C++ code for Newton’s Method: maxPoly.cpp

```cpp
#include <math.h>

double maxPoly(double x0, double coef, double tol){
    // Iterate to convergence
    double x = x0;
    double diff = tol+1;
    double firstDeriv, secondDeriv, xNew;
    while(diff > tol){
        // Compute the first derivative
        firstDeriv = 2*coef*x + 2.3;

        // Compute the second derivative
        secondDeriv = 2*coef;

        // Newton step
        xNew = x - firstDeriv/secondDeriv;

        // Compute difference for convergence check and update
        diff = fabs(xNew - x);
        x = xNew;
    }

    // Function output
    return x;
}
```
Listing 3 shows the file `main.cpp` which corresponds to the Matlab script `main.m` in Listing 1. Two general notes about C++ syntax will be beneficial:

1. Single-line comments in C++ begin with `//`, as opposed to `%` in Matlab. Multi-line comments begin with `/*` and end with `*/`.

2. Functions and conditional statements in C++ begin and end with curly braces `{}`, whereas in Matlab only the end point is explicitly defined with the statement `end`.

The first notable difference between `main.cpp` and `main.m` arises in lines 1 and 3 of the former,

```cpp
#include <Eigen/Dense>
using namespace Eigen;
```

where the Eigen library is called: Eigen (http://eigen.tuxfamily.org) is a template library that provides basic linear algebra functionality. By default, C++ does not load many of the basic libraries that are beneficial for scientific computing – these must be invoked explicitly in the software.

The next difference is the declaration of the function `maxPoly` in line 5

```cpp
double maxPoly(double x0, double coef, double tol);
```

Before any variable or function can be used in C++, it must be declared and initialized. Further, declarations require a statement of type: in this case the `double` preceding the name of the function states that the function will return a double precision variable, and the instances of `double` before each of the function arguments also state that the arguments will be double precision values. The function itself is only declared in `main.cpp` and not defined – the definition is fully enclosed in `maxPoly.cpp`. However, in order to utilize the function, `main.cpp` must have access to the definition of `maxPoly` and not only its declaration. This is accomplished by linking the two C++ files at compile time, which can either be done on the command line or in a separate makefile, a topic which is beyond the scope of this paper. To see how this is accomplished in a makefile, readers can download code for this example at www.parallelecon.com/basic-gpu/.
Unlike Matlab, which allows users to write an interactive script, all C++ code must be
wrapped in an outer function entitled main. This is seen in line 7 of Listing 3. Convention
is that main returns an integer value: 0 if the program is successful, 1 otherwise. Within the
main function, we see the same operations being performed as in main.m. First, the grid of
second-order coefficients, paramGrid, is constructed

```cpp
int nParam = 100000000;
double paramMin = -0.9;
double paramMax = -0.1;
VectorXd paramGrid = VectorXd::LinSpaced(nParam, paramMin, paramMax);
```

Clearly, nParam is declared to be an integer and paramMin and paramMax are double precision.
Less obviously, paramGrid is declared as type VectorXd, which is a double precision vector
made available by Eigen. The function LinSpaced(n,a,b) constructs an equally spaced
array of n values between a and b.

Lines 17-20

```cpp
VectorXd argMaxVals = VectorXd::Zero(nParam);
for(int i = 0 ; i < nParam ; ++i){
    argMaxVals(i) = maxPoly(2.2, paramGrid(i), 0.00001);
}
```

then allocate storage for the arg max values and loop over paramGrid, performing the max-
imization by calling maxPoly for each value of the grid. Aside from previously mentioned
syntactical differences, these lines are identical to their Matlab counterpart. Listings 2 and
4 show that the same is true of the functions maxPoly.m and maxPoly.cpp: aside from
previously mentioned syntactical differences and line 1 of main.cpp

```cpp
#include <math.h>
```

which explicitly invokes the basic math library math.h, these two files are essentially iden-
tical.
2.3.3 CUDA C

CUDA C is a set of C/C++ callable functions that provide an interface to NVIDIA graphics devices. Listings 5 and 6 display parallel GPU code, written in C++, making use of CUDA C function calls. Note that the file name extensions have been changed from .cpp to .cu. The first line of Listing 5

```
#include "maxPoly.cu"
```
serves the purpose of declaring and defining the function in maxPoly.cu. In the C++ example this was accomplished by linking the two files at compile time. However, linking static CUDA C code was not available until very recently (as discussed in Section 5) and so the legacy method of “linking” two files is to explicitly aggregate the source code (through the `#include` command) into a single file. This could also be done in C++, using the command `#include ‘‘maxPoly.cpp’’`.

Lines 7-9 of Listing 5 show that nParam, paramMin and paramMax are declared and initialized exactly as in main.cpp, however the initialization of paramGrid on lines 10 and 11 is somewhat different:

```
double* paramGrid = new double[nParam];
for(int i = 0 ; i < nParam ; ++i)
    paramGrid[i] = paramMin + i*(paramMax-paramMin)/(nParam-1);
```

Where the C++ code declared paramGrid to be an Eigen vector of double precision values and initialized the grid with the function `LinSpaced`, the CUDA C implementation is a bit more rudimentary: it declares a basic C array on line 10 and then initializes each value of the array with a for loop. The reason for this is that the CUDA compiler, nvcc, does not support the object-oriented functionality of the Eigen library and hence cannot compile CUDA code with Eigen references. While it would be possible to separate all CUDA calls in separate files to be compiled by nvcc and still use the Eigen library in main.cu, such maneuvering provides little benefit within the context of this simple problem.

One of the major differences between main.cpp and main.cu centers on the use of host and device memory: in order to maximize the polynomial for each value of paramGrid on
Listing 5: CUDA C code for polynomial maximization problem: main.cu

```c
#include "maxPoly.cu"

int main()
{

  // Grid for order 2 coefficient
  int nParam = 1000000000;
  double paramMin = -0.9;
  double paramMax = -0.1;
  double* paramGrid = new double[nParam];
  for (int i = 0; i < nParam; ++i) paramGrid[i] = paramMin + i*(paramMax-paramMin)/(nParam-1);

  // Copy parameter grid from CPU to GPU memory
  double* paramGridDevice;
  cudaMalloc((void**)&paramGridDevice, nParam*sizeof(double));
  cudaMemcpy(paramGridDevice, paramGrid, nParam*sizeof(double), cudaMemcpyHostToDevice);

  // Storage for argmax values
  double* argMaxValsDevice;
  cudaMalloc((void**)&argMaxValsDevice, nParam*sizeof(double));

  // Maximize for each coefficient
  maxPoly<<<1,256>>>(2.2, paramGridDevice, 0.00001, argMaxValsDevice);

  // Copy argmax values from GPU to CPU memory
  double* argMaxVals = new double[nParam];
  cudaMemcpy(argMaxVals, argMaxValsDevice, nParam*sizeof(double), cudaMemcpyDeviceToHost);

  return 0;
}
```
#include <math.h>

global void maxPoly(double x0, double* coef, double tol, double* argMax)
{
  // Thread ID
  int i = threadIdx.x;

  // Iterate to convergence
  double x = x0;
  double diff = tol + 1;
  double firstDeriv, secondDeriv, xNew;
  while (diff > tol) {
    // Compute the first derivative
    firstDeriv = 2 * coef[i] * x + 2.3;

    // Compute the second derivative
    secondDeriv = 2 * coef[i];

    // Newton step
    xNew = x - firstDeriv / secondDeriv;

    // Compute difference for convergence check and update
    diff = fabs(xNew - x);
    x = xNew;
  }

  // Function output
  argMax[i] = x;
}
the GPU, the grid must first be declared and initialized in host memory, as on lines 10 and 11 of Listing 5, and then transferred to device memory. The transfer is accomplished in two steps. First, on lines 14 and 15

```c
double* paramGridDevice;

cudaMalloc((void**)&paramGridDevice, nParam*sizeof(double));
```

memory is explicitly allocated on the device. The syntax may look strange, but the essential features are that line 14 declares a new double precision vector `paramGridDevice` (in reality, the asterisk states that `paramGridDevice` is a “pointer” which points to a block of memory that has been set aside for double precision variables) and line 15 allocates enough space in memory for `nParam` double precision variables. The second step on line 16

```c
cudaMemcpy(paramGridDevice, paramGrid, nParam*sizeof(double), cudaMemcpyHostToDevice);
```

uses the function `cudaMemcpy` to explicitly copy the variable `paramGrid` in host memory to the empty vector `paramGridDevice` in device memory. Similar syntax is used to declare and initialize a vector `argMaxValsDevice` on lines 19 and 20, but since the initial values are unimportant there is no need to explicitly copy predefined values from host to device memory. Only after the optimization has been performed, with the arg max values stored in `argMaxValsDevice`, does the code return the solution to host memory on lines 26 and 27

```c
double* argMaxVals = new double[nParam];
cudaMemcpy(argMaxVals, argMaxValsDevice, nParam*sizeof(double), cudaMemcpyDeviceToHost);
```

Note that to complete the transfer, the variable `argMaxVals` must first be declared and initialized in host memory, since this was not done previously.

The final, crucial difference between Listings 3 and 5 occurs at line 23, where the loop over `paramGrid` has been eliminated and replaced with a single CUDA C call to the kernel `maxPoly`:

```c
maxPoly<<<1,256>>>(2.2, paramGridDevice, 0.00001, argMaxValsDevice);
```
The \texttt{<<<x,y>>>\> syntax is the core CUDA C interface to request parallel operation on a data structure. The first argument can either be an integer or a one-, two- or three-dimensional object of type \texttt{dim3}, which specifies the dimensions of the grid which contains thread blocks (i.e. the thread blocks can be arranged in a larger grid structure). If this argument is a single integer, as above, it indicates that all thread blocks reside in a single grid element (i.e. the grid is not used). The second argument is either an integer or a one- or two-dimensional object of type \texttt{dim3} which specifies the the dimensions of a thread block. In the example above, a single integer specifies that 256 threads reside in a single-dimensional block. The upshot is that line 23 request the operations in the kernel \texttt{maxPoly} to be performed in parallel on blocks of 256 elements of \texttt{paramGridDevice}. It is important to note that while different block sizes are optimal for different GPUs (depending on the number of processors), this variable does not need to be changed when moving the code from one processor to another – even if a GPU has fewer than 256 cores.

The C++ function \texttt{maxPoly.cpp} and CUDA kernel \texttt{maxPoly.cu} are almost exactly identical. The first difference occurs in the kernel definition on line 3 of Listing 6:

\begin{verbatim}
__global__ void maxPoly(double x0, double* coef, double tol, double* argMax){
\end{verbatim}

The following is a break-down of the how this line differs from the corresponding definition in \texttt{maxPoly.cpp}:

- \texttt{__global__} is CUDA C syntax for declaring a kernel (referring to global device memory).

- The kernel must return type \texttt{void}, which is true of all CUDA kernels (as compared to the \texttt{double} return type of \texttt{maxPoly.cpp}). This means that \texttt{maxPoly.cu} returns nothing.

- The second argument of the kernel is the full vector (in reality, a pointer to the vector in memory) of possible second-order coefficients, rather than a single element of the coefficient array.

- Because it returns \texttt{void}, the kernel has been augmented with an additional argument, \texttt{argMax}, which is an empty vector where the solutions are stored. In particular, since
a pointer to the location of the vector in memory is passed (the * notation) the values can be modified by the function and will remain modified upon exit.

Finally, line 6 of `maxPoly.cu`

```c
int i = threadIdx.x;
```

is the operative line of code that CUDA uses to assign data elements to processor threads: a unique thread ID is assigned to variable i, which is then used to access the respective elements of the parameter grid `coef` on lines 15 and 18, and to assign the final solution to `argMax` on line 30.

In summary, this code allows the CUDA runtime environment to divide thread blocks among available multiprocessors, which then schedules individual threads to individual cores. As emphasized above, the scheduling of blocks is transparent to the user and scales automatically to the number of multiprocessors. Each thread process then accesses a unique ID in the thread block via the `threadIdx` command. The end result is that the sequential loop is eliminated and each GPU core is able to issue the kernel instructions for optimization in parallel on individual elements of the parameter vector. Because of transparent scaling, this code can run on a laptop with only 32 GPU cores or on a Tesla C2075 with 448 cores without modification.

### 2.3.4 Thrust

As mentioned above, Thrust is a free template library that can be called within C/C++ and which provides an alternate interface to GPU hardware. Listings 7 and 8 display parallel GPU code written in C++, making use of the Thrust template library. The primary advantage of Thrust is that it combines the conciseness of Matlab and C++/Eigen code with the ability to schedule parallel work on a GPU. In particular, Thrust eliminates the need for explicit memory allocation and transfer between host and device. Although the transfer must still occur, allocating and copying a data object in device memory is as simple as

```c
double* Y = new double[N]; // Allocate a vector, Y, of N elements in host memory
thrust::device_vector<double> X = Y; // Allocate and copy to device memory
```
Listing 7: Thrust code for polynomial maximization problem: main.cu

```c
#include <thrust/device_vector.h>
#include <thrust/sequence.h>
#include <thrust/transform.h>
#include "maxPoly.hpp"

int main()
{
  int nParam = 1000000000;
  double paramMin = -0.9;
  double paramMax = -0.1;
  thrust::device_vector<double> paramGrid(nParam);
  thrust::sequence(paramGrid.begin(), paramGrid.end(), paramMin, (paramMax-paramMin)/(nParam - 1));

  thrust::device_vector<double> argMaxVals(nParam);
  thrust::transform(paramGrid.begin(), paramGrid.end(), argMaxVals.begin(), maxPoly(2.2, 0.00001));

  return 0;
}
```
#include <math.h>

struct maxPoly{

  // Arguments
  const double x0; ///< Initial value
  const double tol; ///< Convergence criterion

  // Constructor
  maxPoly(const double _x0, const double _tol) : x0(_x0), tol(_tol) {}

  // Kernel
  host_device
double operator()(const double coef) const {
    // Iterate to convergence
    double x = x0;
    double diff = tol+1;
    double firstDeriv, secondDeriv, xNew;
    while(diff > tol){
      // Compute the first derivative
      firstDeriv = 2*coef*x + 2.3;

      // Compute the second derivative
      secondDeriv = 2*coef;

      // Newton step
      xNew = x - firstDeriv/secondDeriv;

      // Compute difference for convergence check and update
      diff = fabs(xNew - x);
      x = xNew;
    }

    // Function output
    return x;
  }

};
in contrast to the excessively verbose use of `cudaMalloc` and `cudaMemcpy` in CUDA C. This greatly facilitates the development of software as it allows the user to work at a high level of abstraction, without the need to deal with the minor details of memory allocation and transfer.

Lines 1-3 of Listing 7 include the relevant Thrust libraries for use in C++ and line 4

```cpp
#include "maxPoly.hpp"
```

is the equivalent of including the `maxPoly` kernel source code, which will be described below. The declarations of `nParam`, `paramMin` and `paramMax` on lines 10-12 are identical to those in C++ and CUDA C, so the first major difference arises on lines 13 and 14 with the declaration and initialization of `paramGrid`:

```cpp
thrust::device_vector<double> paramGrid(nParam);
thrust::sequence(paramGrid.begin(), paramGrid.end(),
paramMin, (paramMax-paramMin)/(nParam-1));
```

The syntax `thrust::device_vector` instantiates a vector directly in device memory. The function `thrust::sequence(start, end, lower, step)` then constructs an equally spaced sequence of points between the positions `start` and `end` of a Thrust vector, beginning at the value `lower`, with the grid points `step` units apart. Unlike CUDA C, where `paramGrid` first had to be computed in host memory and transferred to device memory, we see that Thrust allows functionality to build such a grid directly in the device memory. This not only saves time in transferring data, but also results in more streamlined code.

Line 17 then declares and allocates memory for a device vector, `argMaxVals`, which is not transferred to host memory at the end of the program because Thrust allows users to access and manipulate members of a device vector as if they reside in host memory. The important parallel operation of the file occurs at line 18:

```cpp
thrust::transform(paramGrid.begin(), paramGrid.end(),
argMaxVals.begin(), maxPoly(2.2, 0.00001));
```

In contrast to the `<<<x,y>>>` syntax of CUDA C, the parallel interface in Thrust is provided by two functions: `thrust::transform` and `thrust::for_each`. Specifically,
thrust::transform(inputStart, inputEnd, outputStart, function)

applies function to each element of a Thrust vector between inputStart and inputEnd and places the output in the Thrust vector starting at outputStart. Although it is not described here, thrust::for_each provides similar functionality. Most users find this interface to be a bit more intuitive than that of CUDA.

While we previously saw that the files maxPoly.m, maxPoly.cpp and maxPoly.cu were almost identical, a brief glance at Listing 8 shows substantial differences in the Thrust equivalent, maxPoly.hpp (which is referred to as a “functor” or function object). The major differences arise from the fact that Thrust encloses the “kernel” in a C++ class structure rather than in a simple function. Line 3

    struct maxPoly{

is the C/C++ syntax for declaring such an object and Lines 6, 7 and 10

    const double x0; ///< Initial value
    const double tol; ///< Convergence criterion
    maxPoly(const double _x0, const double _tol) : x0(_x0), tol(_tol) {}

declare the members of the object. In this case, the members are the actual function maxPoly and the arguments to the function (the initial value of the arg max, x0, and the Newton convergence tolerance, tol). Lines 13 and 14

    __host__ __device__
    double operator()(const double coef) const {

provide the esoteric syntax for an operator (), which is the object interface for maxPoly, and the instructions between lines 14 and 35 form the core of maxPoly, which are identical to those found in maxPoly.cpp. Note that there is no explicit use of a thread ID in Thrust, nor does maxPoly take paramGrid as an argument – these details are handled transparently via thrust::transform. Also, where users must specify a thread block structure in CUDA C, Thrust handles details of blocks and grids under the hood.

The final advantage of Thrust is that it has separate backends for CUDA and OpenMP; that is, the same software can access GPU or CPU parallelism (although, not both
at the same time). The result is that Thrust software can run without modification on systems that do not have GPU capabilities, but that have multiple CPUs. This will be demonstrated in the example of the following section. For more details on using the Thrust interface, see Bell and Hoberock (2012) or the Thrust Quickstart Guide (http://code.google.com/p/thrust/wiki/QuickStartGuide).

3 Example: Value Function Iteration

We will now turn our attention to a specific example where parallelism can greatly speed the computation of an economic model. Specifically, we will consider a canonical real business cycle (RBC) model solved with value function iteration (VFI). While this model is simple, it is an excellent illustration of the benefits of a parallel architecture. The results of this section are based upon those of Aldrich et al. (2011) with some minor modifications and code to replicate the results is available at http://www.parallelecon.com/vfi/.

3.1 Model

The economy is populated by a representative agent with preferences over a single consumption good. The agent seeks to maximize expected lifetime utility,

\[ E_0 \left[ \sum_{t=0}^{\infty} \beta^t \frac{C_t^{1-\gamma}}{1-\gamma} \right], \]  

where \( E_0 \) is the conditional expectations operator, \( \beta \) is the discount factor, and \( \gamma \) is the coefficient of relative risk aversion.

The agent receives income each period by renting labor and capital to a representative firm and chooses consumption and investment so as to satisfy the budget constraints

\[ C_t + I_t = w_t + r_t K_t, \quad \forall t, \]  

where \( w_t \) is the wage paid for a unit of labor, \( r_t \) is the rental rate for a unit of capital, \( I_t \) is investment, and where we have assumed that labor, \( L_t \), is normalized to unity and supplied inelastically because it does not enter the agent’s utility function. Capital depreciates each
period according to

\[ K_{t+1} = I_t + (1 - \delta)K_t, \quad \forall t, \]  

(5)

where \( \delta \) is the depreciation rate.

The representative firm produces output according to

\[ Y_t = Z_t K_t^\alpha, \quad \forall t \]  

(6)

where the total factor of productivity (TFP), \( Z_t \), follows the law of motion

\[ \log(Z_t) = \rho \log(Z_{t-1}) + \varepsilon_t, \quad \text{where} \quad \varepsilon_t \sim \mathcal{N}(0, \sigma^2) \quad \forall t. \]  

(7)

Combining Equations (4)-(6) with the market clearing condition \( C_t = Y_t \), we arrive at the aggregate resource constraint

\[ K_{t+1} + C_T = Z_t K_t^\alpha + (1 - \delta)K_t, \quad \forall t. \]  

(8)

Since the welfare theorems are satisfied, we can find the agent’s optimal consumption path by solving a recursive version of a social planner’s problem

\[ V(K, Z) = \max_c \left\{ C^{1-\gamma} \frac{1}{1 - \gamma} + \beta \mathbb{E} [V(K', Z') | Z] \right\} \]  

subject to

\[ K' = ZK^\alpha + (1 - \delta)K - C. \]  

(9)

(10)

It is not requisite to solve the model with VFI in order to achieve the advantages of massive parallelism - we could obtain similar benefits by working directly with the equilibrium conditions. However, the problem is most easily illustrated in the present format.

3.2 Algorithm

The basic VFI solution algorithm follows.

1: Fix some \( \tau > 0 \) which will determine convergence and set \( \varepsilon = \tau + 1 \).
Compute the deterministic steady-state level of capital, $K_{ss}$, and set $K = 0.95K_{ss}$ and $\bar{K} = 1.05K_{ss}$. Discretize the state space for capital so that it is confined to a grid of $N_k$ equally-spaced values between $K$ and $\bar{K}$. Denote the grid by $\mathcal{K}$.

Use the method of Tauchen (1986) to discretize the state space for the log of TFP so that it is confined to a grid of $N_z$ equally-spaced values between $z$ and $\bar{z}$ (where $z = \log(Z)$). Denote the grid for TFP levels by $\mathcal{Z}$ and the matrix of transition probabilities $P$, where the probability of transitioning from $Z$ to $Z'$ is expressed as $P(Z,Z')$.

Guess initial values of the value function, $V^0$, for each pair of possible values of the state variables, $K$ and $Z$ (i.e. $V^0$ is an $N_k \times N_z$ matrix). In particular, set $V^0$ to be equal to the deterministic steady-state values of the value function.

**while $\varepsilon > \tau$ do**

for each $K \in \mathcal{K}$

for each $Z \in \mathcal{Z}$

for each $K' \in \mathcal{K}$

Compute

$$C(K, Z, K') = ZK^\alpha + (1 - \delta)K - K'$$

$$Exp(K, Z, K') = \sum_{Z' \in \mathcal{Z}} V^0(K', Z') \ast P(Z, Z')$$

$$\tilde{V}(K, Z, K') = \frac{C(K, Z, K')^{1-\gamma}}{1 - \gamma} + Exp(K, Z, K').$$

end for

Set

$$V(K, Z) = \max_{K'} \tilde{V}(K, Z, K').$$

end for

end for

Compute the difference between the updated value function and $V^0$:

$$\varepsilon = ||V - V^0||_\infty.$$
end while

This algorithm is very straightforward, and variations of it are typically taught in first-year graduate macroeconomics courses. A basic implementation would involve computing the quantities in Equations (11) - (13) in a serial fashion for each value of $K' \in \mathcal{K}$ in the loop at Step 8. If either $\mathcal{K}$ or $\mathcal{Z}$ is a very dense grid, Step 8 may involve many thousands of serial calculations for each of the values in the loops at Steps 6 and 7.

Alternatively, with many parallel processors available, the loops at Steps 6 and 7 could be eliminated and the sequence of instructions nested in Step 8 could be assigned to an individual processor and computed in parallel for each pair $(K, Z)$. To be specific, the suggested parallelism occurs over the loops in Steps 6 and 7, and not the loop in Step 8. While parallelism is possible over the latter loop, it does not prove computationally beneficial for this particular problem (as will be discussed in the next section). The reason that parallelism can be exploited in this problem is that the computations nested within Steps 6 and 7 depend only on the concurrent $(K, Z)$ and not on other values in $\mathcal{K}$ and $\mathcal{Z}$. Aldrich et al. (2011) implement this algorithm and Section 3.3 reports updated results from that paper.

A few algorithmic considerations are important to note. First, the choice of maximization procedure in Equation (14) can have important implications for parallel efficiency. The results of the next section report two maximization methods:

1. A naive grid search, computing $\tilde{V}(K, Z, K')$ for each $K' \in \mathcal{K}$.

2. A binary search, which exploits concavity of the value function to iteratively bisect $\mathcal{K}$ until an optimal value is found (see p. 26 of Heer and Maussner (2005) for an explicit algorithm).

Additionally, there are two basic ways to accelerate the algorithm:

1. Exploiting monotonicity of the value function by only maximizing over $K' \in \tilde{\mathcal{K}}$, where $\tilde{\mathcal{K}}$ is bounded below by $\tilde{K} = \max \{ K, G(K^*, Z) \}$, and where $G(K, Z)$ is the concurrent policy function and $K^* = \max \{ \tilde{K} \in \mathcal{K} : \tilde{K} < K \}$.

2. Policy function iteration: for some number $N$, perform the maximization in Equation (14) at every $N$th iteration. Otherwise, choose $V(K, Z) = \tilde{V}(K, Z, G(K, Z))$. 

30
Unfortunately, policy function iteration does not preserve concavity of the value function, and so is not compatible with binary search; for this reason it is only used with the grid search method for the results reported in Section 3.3. In addition, it is clear that exploiting monotonicity introduces dependencies across computations for pairs of \((K, Z)\) and so cannot be used with parallel versions of the software; for this reason, only single-threaded, sequential versions of the algorithm make use of monotonicity.

### 3.3 Results

Table 1 reports calibrated parameter values for the model. These values are standard in the literature for a quarterly calibration of a basic RBC model.

<table>
<thead>
<tr>
<th>(\beta)</th>
<th>(\gamma)</th>
<th>(\alpha)</th>
<th>(\delta)</th>
<th>(\rho)</th>
<th>(\sigma)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.984</td>
<td>2</td>
<td>0.35</td>
<td>0.01</td>
<td>0.95</td>
<td>0.005</td>
</tr>
</tbody>
</table>

**Table 1:** Model calibration

All solutions were computed in double precision (which is less favorable for the GPU computations since current GPUs are relatively faster at single-precision arithmetic), with a convergence criterion of \(\tau = (1 - \beta)1e - 8\). The grid for TFP was discretized over four values using the method of Tauchen (1986). The grid for capital was discretized with increasing density in order to assess the performance of massively parallel techniques over an increasingly complex problem. When making use of policy function iteration, maximization was performed at every 20th iteration.

Tables 2 and 3 report timing results for various software implementations of the algorithm in Section 3.2. The methods include

- Single-threaded, sequential C++, making use of the Eigen template library for linear algebra computations.
- Single-threaded, sequential Matlab. This is done to compare with what the majority of economists would use to solve the problem.
• Thrust, using the OpenMP backend to solve the problem in parallel on several CPU cores.

• Thrust, using the CUDA backend to solve the problem in parallel on the GPU.

• CUDA C.

Recall that the GPU version of Thrust and CUDA C both make use of the CUDA architecture, but that they provide different interfaces. In all cases, the ratios in the tables are relative to the single-threaded C++ times. Table 2 corresponds to the binary search maximization method and Table 3 corresponds to the grid search maximization method. All results were obtained on a on a 4U rackmount server with a single quad-core Intel Xeon 2.4 GHz CPU and two NVIDIA Tesla C2075 GPUs, although only one of the GPUs was used for the Thrust/CUDA and CUDA C timing results. The Thrust/OpenMP software, however, made use of all four of the CPU cores.

<table>
<thead>
<tr>
<th>$N_k$</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1,024</th>
<th>2,048</th>
<th>4,096</th>
<th>8,192</th>
<th>16,384</th>
<th>32,768</th>
<th>65,536</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPP</td>
<td>0.547</td>
<td>1.35</td>
<td>3.41</td>
<td>9.05</td>
<td>25.73</td>
<td>84.58</td>
<td>297.32</td>
<td>1,114.95</td>
<td>4,653.81</td>
<td>19,421.90</td>
</tr>
<tr>
<td>Matlab</td>
<td>48.97</td>
<td>98.36</td>
<td>263.18</td>
<td>426.57</td>
<td>920.10</td>
<td>2,077.24</td>
<td>5,020.26</td>
<td>16,129.32</td>
<td>45,070.47</td>
<td>140,341.10</td>
</tr>
<tr>
<td>Matlab Ratio</td>
<td>89.55</td>
<td>72.67</td>
<td>59.50</td>
<td>47.13</td>
<td>35.76</td>
<td>24.56</td>
<td>16.89</td>
<td>14.47</td>
<td>9.68</td>
<td>7.23</td>
</tr>
<tr>
<td>Thrust/OpenMP</td>
<td>0.118</td>
<td>0.241</td>
<td>0.519</td>
<td>1.10</td>
<td>2.37</td>
<td>5.04</td>
<td>10.81</td>
<td>23.10</td>
<td>49.53</td>
<td>106.66</td>
</tr>
<tr>
<td>Thrust/OpenMP Ratio</td>
<td>0.217</td>
<td>0.178</td>
<td>0.152</td>
<td>0.121</td>
<td>0.0920</td>
<td>0.0595</td>
<td>0.0364</td>
<td>0.0207</td>
<td>0.0106</td>
<td>0.00549</td>
</tr>
<tr>
<td>Thrust/CUDA Start</td>
<td>6.75</td>
<td>6.70</td>
<td>6.74</td>
<td>6.64</td>
<td>6.62</td>
<td>6.72</td>
<td>6.72</td>
<td>6.75</td>
<td>6.66</td>
<td>6.75</td>
</tr>
<tr>
<td>Thrust/CUDA Solution</td>
<td>0.240</td>
<td>0.273</td>
<td>0.322</td>
<td>0.392</td>
<td>0.711</td>
<td>1.20</td>
<td>2.13</td>
<td>4.26</td>
<td>8.52</td>
<td>17.21</td>
</tr>
<tr>
<td>Thrust/CUDA Total</td>
<td>6.99</td>
<td>6.97</td>
<td>7.06</td>
<td>7.03</td>
<td>7.43</td>
<td>7.82</td>
<td>8.84</td>
<td>11.01</td>
<td>15.19</td>
<td>23.96</td>
</tr>
<tr>
<td>Thrust/CUDA Solution Ratio</td>
<td>0.439</td>
<td>0.292</td>
<td>0.0943</td>
<td>0.0433</td>
<td>0.0276</td>
<td>0.0141</td>
<td>0.00715</td>
<td>0.00382</td>
<td>0.00183</td>
<td>0.000886</td>
</tr>
<tr>
<td>Thrust/CUDA Total Ratio</td>
<td>12.78</td>
<td>5.15</td>
<td>2.07</td>
<td>0.777</td>
<td>0.289</td>
<td>0.0924</td>
<td>0.0297</td>
<td>0.00987</td>
<td>0.00326</td>
<td>0.00123</td>
</tr>
<tr>
<td>CUDA C Solution</td>
<td>0.144</td>
<td>0.156</td>
<td>0.258</td>
<td>0.416</td>
<td>0.731</td>
<td>1.20</td>
<td>2.13</td>
<td>4.26</td>
<td>8.52</td>
<td>17.21</td>
</tr>
<tr>
<td>CUDA C Total</td>
<td>7.12</td>
<td>7.11</td>
<td>7.16</td>
<td>7.38</td>
<td>7.67</td>
<td>8.47</td>
<td>10.05</td>
<td>13.42</td>
<td>20.76</td>
<td>36.19</td>
</tr>
<tr>
<td>CUDA C Solution Ratio</td>
<td>0.263</td>
<td>0.115</td>
<td>0.076</td>
<td>0.0460</td>
<td>0.0284</td>
<td>0.0178</td>
<td>0.0104</td>
<td>0.00581</td>
<td>0.00297</td>
<td>0.00151</td>
</tr>
<tr>
<td>CUDA C Total Ratio</td>
<td>13.01</td>
<td>5.25</td>
<td>2.10</td>
<td>0.816</td>
<td>0.298</td>
<td>0.100</td>
<td>0.0338</td>
<td>0.0120</td>
<td>0.00446</td>
<td>0.00186</td>
</tr>
</tbody>
</table>

Table 2: Timing results (in seconds) for the RBC/VFI problem using binary search maximization.

Table 2 demonstrates the great benefits of parallelism for the VFI problem. Most notably, as the capital grid density increases, the GPU implementations become increasingly fast.
Table 3: Timing results (in seconds) for the RBC/VFI problem using grid search maximization.

<table>
<thead>
<tr>
<th>(N_k)</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1,024</th>
<th>2,048</th>
<th>4,096</th>
<th>8,192</th>
<th>16,384</th>
<th>32,768</th>
<th>65,536</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPP</td>
<td>0.137</td>
<td>0.332</td>
<td>0.900</td>
<td>2.73</td>
<td>9.17</td>
<td>33.40</td>
<td>126.72</td>
<td>496.24</td>
<td>1,977.93</td>
<td>7,892.46</td>
</tr>
<tr>
<td>Matlab</td>
<td>11.18</td>
<td>22.43</td>
<td>45.23</td>
<td>93.67</td>
<td>204.50</td>
<td>476.54</td>
<td>1,212.05</td>
<td>3,587.33</td>
<td>11,189.66</td>
<td>37,720.38</td>
</tr>
<tr>
<td>Matlab Ratio</td>
<td>81.42</td>
<td>67.59</td>
<td>50.26</td>
<td>34.28</td>
<td>22.29</td>
<td>14.27</td>
<td>9.56</td>
<td>7.23</td>
<td>5.66</td>
<td>4.78</td>
</tr>
<tr>
<td>Thrust/OpenMP</td>
<td>0.0607</td>
<td>0.156</td>
<td>0.490</td>
<td>1.70</td>
<td>6.37</td>
<td>24.64</td>
<td>96.95</td>
<td>383.30</td>
<td>1,524.40</td>
<td>6,081.77</td>
</tr>
<tr>
<td>Thrust/OpenMP Ratio</td>
<td>0.442</td>
<td>0.471</td>
<td>0.544</td>
<td>0.628</td>
<td>0.694</td>
<td>0.738</td>
<td>0.765</td>
<td>0.772</td>
<td>0.771</td>
<td>0.771</td>
</tr>
<tr>
<td>Thrust/CUDA Start</td>
<td>6.73</td>
<td>6.73</td>
<td>6.69</td>
<td>6.66</td>
<td>6.72</td>
<td>6.78</td>
<td>6.77</td>
<td>6.75</td>
<td>6.74</td>
<td>6.78</td>
</tr>
<tr>
<td>Thrust/CUDA Solution</td>
<td>0.173</td>
<td>0.234</td>
<td>0.348</td>
<td>0.628</td>
<td>1.53</td>
<td>4.23</td>
<td>13.71</td>
<td>53.33</td>
<td>191.99</td>
<td>774.87</td>
</tr>
<tr>
<td>Thrust/CUDA Total</td>
<td>6.90</td>
<td>6.97</td>
<td>7.04</td>
<td>7.29</td>
<td>8.26</td>
<td>11.02</td>
<td>20.48</td>
<td>60.08</td>
<td>198.73</td>
<td>781.65</td>
</tr>
<tr>
<td>Thrust/CUDA Solution Ratio</td>
<td>1.26</td>
<td>0.704</td>
<td>0.387</td>
<td>0.230</td>
<td>0.167</td>
<td>0.127</td>
<td>0.108</td>
<td>0.107</td>
<td>0.0971</td>
<td>0.0992</td>
</tr>
<tr>
<td>Thrust/CUDA Total Ratio</td>
<td>50.26</td>
<td>20.99</td>
<td>7.83</td>
<td>2.67</td>
<td>0.900</td>
<td>0.330</td>
<td>0.162</td>
<td>0.121</td>
<td>0.100</td>
<td>0.0990</td>
</tr>
<tr>
<td>CUDA C Solution</td>
<td>0.103</td>
<td>0.134</td>
<td>0.257</td>
<td>0.664</td>
<td>2.12</td>
<td>7.81</td>
<td>29.76</td>
<td>116.80</td>
<td>462.31</td>
<td>1,844.37</td>
</tr>
<tr>
<td>CUDA C Total</td>
<td>7.04</td>
<td>7.07</td>
<td>7.20</td>
<td>7.62</td>
<td>8.93</td>
<td>14.71</td>
<td>36.68</td>
<td>123.65</td>
<td>469.28</td>
<td>1,851.36</td>
</tr>
<tr>
<td>CUDA C Solution Ratio</td>
<td>0.750</td>
<td>0.404</td>
<td>0.286</td>
<td>0.243</td>
<td>0.231</td>
<td>0.234</td>
<td>0.235</td>
<td>0.234</td>
<td>0.234</td>
<td>0.234</td>
</tr>
<tr>
<td>CUDA C Total Ratio</td>
<td>51.26</td>
<td>21.30</td>
<td>8.01</td>
<td>2.79</td>
<td>0.974</td>
<td>0.440</td>
<td>0.289</td>
<td>0.249</td>
<td>0.237</td>
<td>0.235</td>
</tr>
</tbody>
</table>

relative to the C++/CPU times, where at the largest grid size considered (\(N_k = 65,536\)) total Thrust/CUDA and CUDA C times are 810 times and 537 times faster, respectively, than C++. Interestingly, for each of these implementations there is an overhead cost for initializing the CUDA runtime environment that corresponds to just less than 7 seconds. This overhead costs swamps the total computation times for small grid sizes and results in the total GPU times only improving upon serial C++ times for \(N_k \geq 1024\). It is also interesting to note that Thrust/CUDA is uniformly more efficient than straight CUDA C. This is most likely a result of the block sizes not being optimally chosen for the CUDA C implementation, which is suggestive of the advantage of using a high-level library like Thrust which automatically tunes a variety of parameters for the specific hardware.

Perhaps most surprising is the performance of Thrust/OpenMP. Although this implementation only utilizes four CPU cores, it improves upon the single-threaded C++ times by more than a factor of four for \(N_K \geq 128\). In fact for \(N_k = 65,536\), Thrust/OpenMP is 182 times faster than C++. This is most likely due to efficiencies in matrix operations in the Thrust library relative to the Eigen library. While it is possible to link the Intel MKL library
to Eigen, this implementation did not make use of MKL - linking the two would improve the C++ times substantially. In addition, since Thrust/OpenMP incurs no overhead start-up cost, it outperforms the GPU implementations for small grid sizes.

Table 3 presents much more moderate results for parallelization: for the largest Grid, Thrust/CUDA and CUDA C only 10 and 4 times faster, respectively, than C++. This is a result of the fact that when $K$ is large, the number of sequential operations performed on each GPU core in Step 8 of the algorithm in Section 3.2 likewise becomes very large - on the order of $N_k$ total operations. The upshot is that the naive grid search treats the GPU cores like CPU cores - requesting many serial operations - something for which they are not optimized. Heer and Maussner (2005) (p.26) notes that binary search, on the other hand, requires only a maximum of $\log_2(N_k)$ operations in Step 8 to achieve the optimum. For the grid sizes in this paper, this amounts to between 7 and 16 sequential operations on each GPU core. For this reason, binary search is much better suited for massively parallel architectures.

Thrust/OpenMP also performs relatively worse for the grid search method, yielding times that are only 10 times faster than C++ for the largest grids. As before, the gain in excess of $4\times$ is most likely attributed to the relative efficiency of matrix operations, but is less than the total gain with binary search because the maximization step occurs much less frequently (due to policy function iteration).

Comparing Tables 2 and 3, we see that the C++ code is much faster for grid search maximization. While the number of sequential operations for grid search is far greater than binary search, the former employs policy function iteration to accelerate computation. It is clear from the tables that the savings afforded by policy function iteration dominate the reduced cost of computations using binary search in C++. Table 4 reports the ratios of Thrust/OpenMP, Thrust/CUDA and CUDA C times using binary search relative to C++ times using grid search. Attributing the best solution times to each implementation results in Thrust/OpenMP, Thrust/CUDA and CUDA C being 74 times, 333 times and 218 times faster, respectively, than C++. 

34
Table 4: Ratios of total computation times for the RBC/VFI problem using grid search maximization for C++ and binary search maximization for the other implementations.

<table>
<thead>
<tr>
<th>$N_k$</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1,024</th>
<th>2,048</th>
<th>4,096</th>
<th>8,192</th>
<th>16,384</th>
<th>32,768</th>
<th>65,536</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thrust/OpenMP Ratio</td>
<td>0.863</td>
<td>0.725</td>
<td>0.577</td>
<td>0.402</td>
<td>0.258</td>
<td>0.151</td>
<td>0.0853</td>
<td>0.0465</td>
<td>0.0250</td>
<td>0.0135</td>
</tr>
<tr>
<td>Thrust/CUDA Total Ratio</td>
<td>50.87</td>
<td>21.01</td>
<td>7.85</td>
<td>2.57</td>
<td>0.810</td>
<td>0.234</td>
<td>0.0698</td>
<td>0.0222</td>
<td>0.00768</td>
<td>0.00304</td>
</tr>
<tr>
<td>CUDA C Total Ratio</td>
<td>51.82</td>
<td>21.42</td>
<td>7.96</td>
<td>2.70</td>
<td>0.836</td>
<td>0.254</td>
<td>0.0793</td>
<td>0.0271</td>
<td>0.0105</td>
<td>0.00459</td>
</tr>
</tbody>
</table>

4 Example: Generalized Stochastic Simulation Algorithm

The Generalized Stochastic Simulation Algorithm (GSSA) of Judd et al. (2011b) is a second example that highlights the potential for parallelism in economic computing. Within the specific context of a multi-country real business cycle model, the GSSA algorithm also illustrates limitations of parallelism.

4.1 The Basic Algorithm

As the names suggests, GSSA is a solution method for economic models that limits attention to an ergodic subset of the state space via stochastic simulations. For this reason, the method can accommodate a large number of state variables, unlike other global methods, such as projection. Judd et al. (2011b) outline the algorithm in the context of the representative agent stochastic growth model presented above in Section 3. Generalizing the utility and production functions of that section to be $u(\cdot)$ and $f(\cdot)$, the Euler equation of the agent’s problem is

$$u'(C_t) = \mathbb{E}_t [\beta u'(C_{t+1})(1 - \delta + Z_{t+1}f'(K_{t+1}))],$$

(16)

where $u(\cdot)$ and $f(\cdot)$ are twice continuously differentiable, strictly increasing, strictly concave and satisfy the first Inada condition, $\lim_{x \to 0} g'(x) = \infty$ for $g = \{u, f\}$. In addition, $f(\cdot)$ is homogeneous of degree one and satisfies the second Inada condition, $\lim_{x \to \infty} f'(x) = 0$. 

35
Since $K_{t+1}$ is time $t$-measurable, Equation (16) can be rewritten as

$$K_{t+1} = \mathbb{E}_t \left[ \beta \frac{u'(C_{t+1})}{u'(C_t)} (1 - \delta + Z_{t+1} f'(K_{t+1})) K_{t+1} \right],$$

which expresses $K_{t+1}$ as a fixed point. GSSA finds an approximate solution to the capital policy function, $K_{t+1} = \mathcal{K}(K_t, Z_t)$, by guessing a policy, simulating a time path for capital, computing the expectations on the RHS of Equation (17) under the assumed policy and using the simulation to update the approximation. This procedure is iterated to convergence.

Let the policy approximation be some flexible functional form denoted by $\Psi(K_t, Z_t; \theta)$, for parameter vector $\theta$. The algorithm follows.

1: Set $i = 1$, choose an initial parameter vector $\theta^{(1)}$ and an initial state $(K_0, Z_0)$.
2: Select a simulation length $T$, draw a sequence of shocks $\{\varepsilon_t\}_{t=1}^T$ and compute $\{Z_t\}_{t=1}^T$ according to Equation (7). Also select a set of $J$ integration nodes $\{\epsilon_j\}_{j=1}^J$ and weights $\{\omega_j\}_{j=1}^J$.
3: Choose a convergence tolerance $\tau$ and set $\bar{\omega} = \tau + 1$.
4: while $\bar{\omega} > \tau$ do
5: for $t = 1, \ldots, T$
6: Compute

$$K_{t+1} = \Psi(K_t, Z_t; \theta^{(i)}),$$

$$C_t = (1 - \delta)K_t + Z_tf(K_t) - K_{t+1}.$$ (18)

7: for $j = 1, \ldots, J$
8: Compute

$$Z_{t+1,j} = Z_t^\rho \exp(\epsilon_j),$$

$$K_{t+2,j} = \Psi \left( K_{t+1}, Z_{t+1,j}; \theta^{(i)} \right),$$

$$C_{t+1,j} = (1 - \delta)K_{t+1} + Z_{t+1,j} f(K_{t+1}) - K_{t+2,j}.$$ (21)

9: end for
10: Compute

$$y_t = \sum_{j=1}^J \left\{ \omega_j \left[ \beta \frac{u'(C_{t+1,j})}{u'(C_t)} (1 - \delta + Z_{t+1,j} f'(K_{t+1})) K_{t+1} \right] \right\}. $$ (23)
11:  \textbf{end for}

12:  Find $\hat{\theta}$ that minimizes the errors $\varepsilon_t$ in the regression equation

$$y_t = \Psi(K_t, Z_t; \theta) + \varepsilon_t$$  \hfill (24)

according to some norm $\| \cdot \|$.

13:  Compute

$$\frac{1}{T} \sum_{t=1}^{T} \left| \frac{K_{t+1}^{(i)} - K_{t+1}^{(i-1)}}{K_{t+1}^{(i)}} \right| = \bar{\omega},$$  \hfill (25)

and

$$\theta^{(i+1)} = (1 - \xi)\theta^{(i)} + \xi\hat{\theta},$$  \hfill (26)

where $\{K_{t+1}^{(i)}\}_{t=1}^{T}$ and $\{K_{t+1}^{(i-1)}\}_{t=1}^{T}$ are the simulated capital values in iterations $i$ and $i-1$ and where $\xi \in (0, 1]$ is a damping parameter.

14:  Set $i = i + 1$.

15:  \textbf{end while}

\textbf{Judd et al. (2011b)} includes a second stage of the algorithm which conducts a stringency test and potentially updates the functional form $\Psi$, increases the simulation length, improves the integration method or imposes a more demanding norm in Equation (24). The second stage is omitted in the exposition below since it has very little bearing on parallelism of the solution method.

A number of simple tasks in GSSA can be outsourced to a massively parallel architecture, such as random number generation in Step 2 or matrix operations in Step 12. However, one of the most time intensive tasks of the algorithm is the computation of expectations in Equation (23) for each $t$ in the loop at Step 5. Fortunately, while the time series of capital in Equation (18) must be computed sequentially, once it is determined the other computations within the simulation loop at Step 5 can be performed in parallel. That is, for a given capital simulation, individual processors can be assigned the task of evaluating Equation (19) and Steps 7 – 10. This is possible, because conditional on $K_{t+1}$, the values of consumption and the integral in Equations (19) and (23) are independent across time periods, $t$. Since GSSA typically involves a large number of simulated points and a low number of integration nodes
and weights, this problem is well suited for a GPU: few, identical operations performed in parallel over a large array of data values (simulated capital). The next section will demonstrate a massively parallel application of GSSA to a multi-country RBC model.

4.2 Multi-country RBC Model

Judd et al. (2011b) use GSSA to solve a multi-country RBC model in order to illustrate the performance of the algorithm as state-space dimensionality grows very large. The model is similar to the representative agent growth model above and in Section 3, however, in the multi-country model each country is hit by a technology shock that is comprised of both worldwide and idiosyncratic shocks at each time period. This results in country-specific levels of capital and technology, all of which are state variables for the policy functions of other countries. In particular, with $N$ countries we can write the planner’s problem as

$$
\max_{(C_t^h,K_{t+1}^h)_{h=1}^{N}, t=0,\ldots,\infty} \mathbb{E}_0 \left[ \sum_{h=1}^{N} \lambda^h \left( \sum_{t=1}^{\infty} \beta^t u^h(C_t^h) \right) \right]
$$

subject to

$$
\sum_{h=1}^{N} C_t^h + \sum_{h=1}^{N} K_{t+1}^h = (1 - \delta) \sum_{h=1}^{N} K_t^h + \sum_{h=1}^{N} Z_t^h f^h(K_t^h) \quad \text{(28)}
$$

$$
\log(Z_{t+1}^h) = \rho \log(Z_t^h) + \varepsilon_{t+1}^h, \quad h = 1,\ldots,N, \quad \text{(29)}
$$

where \(\{\lambda^h\}_{h=1}^{N}\) are welfare weights, \(\{K_0^h, Z_0^h\}_{h=1}^{N}\) are given exogenously and \((\varepsilon_{t+1}^1, \ldots, \varepsilon_{t+1}^N)^\top \sim \mathcal{N}(0_N, \Sigma)\), where

$$
\Sigma = \begin{pmatrix}
2\sigma^2 & \ldots & \sigma^2 \\
\ldots & \ldots & \ldots \\
\sigma^2 & \ldots & 2\sigma^2
\end{pmatrix}. \quad \text{(30)}
$$

The country specific utility and production functions, \(u^h\) and \(f^h\), satisfy the same properties as in the representative agent model. If we assume that all countries have identical preferences and technology, \(u^h = u\) and \(f^h = f\ \forall h\), the optimal consumption profiles are symmetric and the planner selects \(\lambda^h = 1, \forall h\). The resulting optimality conditions are

$$
K_{t+1}^h = \mathbb{E}_t \left[ \beta \frac{u'(C_{t+1}^h)}{u'(C_t^h)} (1 - \delta + Z_{t+1}^h f'(K_{t+1}^h)) K_{t+1}^h \right], \quad h = 1,\ldots,N. \quad \text{(31)}
$$
The solution of system (31) is now characterized by $N$ policy functions $K^h\left(\{K^h_t, Z^h_t\}_{h=1}^N\right)$ for $h = 1, \ldots, N$. It is important to note that each country’s policy function not only depends on domestic capital and TFP, but of all other countries as well. This creates important dependencies that are discussed below.

To solve the model, we now approximate the policy functions with flexible functional forms $\Psi^h(\{K^h_t, Z^h_t\}_{h=1}^N; \theta^h)$, $h = 1, \ldots, N$. The modified GSSA algorithm follows.

1: Set $i = 1$, choose initial parameter vectors $\{\theta^{(1),h}\}_{h=1}^N$ and initial states $\{(K_0, Z_0)\}_{h=1}^N$.

Collect the parameter vectors in a matrix, $\Theta^{(1)} = [\theta^{(1), 1}, \ldots, \theta^{(1), N}]$.

2: Select a simulation length $T$, draw $N$ sequences of shocks $\{\epsilon^h_t\}_{t=1}^T$, $h = 1, \ldots, N$, and compute $\{Z^h_t\}_{t=1}^T$, $h = 1, \ldots, N$, according to Equation (29). Also select a set of $J \times N$ integration nodes $\{\epsilon^h_j\}_{j=1}^J$, $\{\omega^h_j\}_{j=1}^J$, and weights $\{\omega^h_j\}_{j=1}^J$.

3: Choose a convergence tolerance $\tau$ and set $\bar{\omega} = \tau + 1$.

4: while $\bar{\omega} > \tau$ do

5: for $t = 1, \ldots, T$

6: for $h = 1, \ldots, N$

7: Compute

$$K_{t+1}^h = \Psi^h(\{K^h_t, Z^h_t\}_{h=1}^N; \theta^{(i), h}),$$

(32)

8: end for

9: Compute average consumption

$$\bar{C}_t = \frac{1}{N} \sum_{h=1}^N [(1 - \delta)K^h_t + Z^h_t f(K^h_t) - K^h_{t+1}].$$

(33)

10: for $j = 1, \ldots, J$

11: for $h = 1, \ldots, N$

12: Compute

$$Z^h_{t+1,j} = (Z^h_t)^\rho \exp(\epsilon^h_j),$$

(34)

13: end for

14: for $h = 1, \ldots, N$

39
Compute

\[ K_{t+2,j}^h = \Psi^h \left( \{ K_{t+1,j}^h, Z_{t+1,j}^h \}_{h=1}^N; \theta^{(i),h} \right). \] (35)

end for

Compute average consumption

\[ \overline{C}_{t+1,j} = \frac{1}{N} \sum_{h=1}^N [ (1 - \delta) K_{t+1,j}^h + Z_{t+1,j}^h f(K_{t+1,j}^h) - K_{t+2,j}^h ] . \] (36)

end for

for \( h = 1, \ldots, N \)

Compute

\[ y_t^h = \sum_{j=1}^d \left\{ \omega_j^h \left( \beta \frac{u'(\overline{C}_{t+1,j})}{u'(C_t)} [1 - \delta + Z_{t+1,j}^h f(K_{t+1,j}^h) K_{t+1,j}^h] \right) \right\} . \] (37)

end for

end for

for \( h = 1, \ldots, N \)

Find \( \hat{\theta}_h \) that minimizes the errors \( \varepsilon_t^h \) in the regression equation

\[ y_t^h = \Psi^h \left( \{ K_t^h, Z_t^h \}_{h=1}^N; \theta^h \right) + \varepsilon_t^h \] (38)

according to some norm \( \| \cdot \| \).

Compute

\[ \frac{1}{T} \sum_{t=1}^T \left| \frac{K_{t+1,i}^{(i),h} - K_{t+1,i}^{(i-1),h}}{K_{t+1,i}^{(i),h}} \right| = \bar{\omega}^h , \] (39)

where \( \{ K_{t+1,i}^{(i),h} \}_{t=1}^T \) and \( \{ K_{t+1,i}^{(i-1),h} \}_{t=1}^T \) are the simulated capital values in iterations \( i \) and \( i - 1 \) and where \( \xi \in (0, 1] \) is a damping parameter.

end for

Set \( \bar{\omega} = \max_h \{ \bar{\omega}^h \} \) and compute

\[ \Theta^{(i+1)} = (1 - \xi)\Theta^{(i)} + \xi \hat{\Theta} . \] (40)

Set \( i = i + 1 \).
The primary differences between this and the algorithm of Section 4.1 are the loops over countries in Steps 6, 11, 14, 19 and 23, where the previous procedure of simulating a time series for capital, computing expectations and regressing the simulation on values dictated by the policy function is now performed $N$ times. Ideally, the loop over individual countries would only occur once in the algorithm, at an early step, either immediately nesting the loop over time in Step 5 or nested immediately under that loop. However, the cross-sectional state variable dependence inhibits the computations over countries: the values of Equation (32) must be computed $\forall N$ before computing the values in Equation (33). Likewise, the values of Equations (32) and (34) must be computed for all $N$ prior to the values in Equation (35), which must also be computed for all $N$ prior to the values in Equation (37) due to the dependence of $C_{t+1,j}$ on all $N$ values.

If a single loop over $N$ countries were possible, the $TN$ values of Equation (37) could be computed independently across many processing cores. However, in the presence of cross-sectional state variable dependence, such parallelization would result in the simultaneous computation of the objects in Steps 12-19 – an unnecessary replication of work that only needs to be computed once for each $h$ in Step 11. To avoid such duplication, an alternative would be to perform the work within the time loop of Step 5 in parallel. This reduces the scope of parallelism (the number of data objects for which parallel instructions are performed) and increases the complexity of the operations performed on each parallel data object. These are limitations of the problem that inhibit the returns to massive parallelism. It is crucial to note, however, that these limitations are not general features of the GSSA algorithm, but particular issues that arise in the GSSA solution of the multi-country RBC model.

A final alternative is available for parallel computation of the multi-country RBC model. CUDA C and OpenCL allow for synchronization of work that is performed by threads in a single block. That is, by moving the data elements associated with the threads in a block to faster-access shared memory on the multiprocessor, individual threads can perform computations while simultaneously having access to the work of other threads within the block. Forcing synchronization at certain points of the algorithm allows individual cores to work individually while also maintaining access to other computations that will be requisite.
for later steps in the algorithm. In the present example, rather than having each processing core perform the \( N \) computations of Steps 6, 11, 14 and 19, threads within a block could compute those values in parallel and then synchronize at Steps 9 and 17, before independently computing the \( TN \) values of Equation (37). This functionality is not available in higher-level libraries such as Thrust and requires a greater degree of effort in programming (directly in CUDA C or OpenCL). The results below were computed with Thrust and so do not make use of thread synchronization. Hence, it is important to keep in mind that with a greater degree of programming effort, the returns to parallelization of the GSSA multi-country solution could be improved, perhaps substantially.

Table 5 reports solution times for the multi-country model under a variety of configurations considered in Judd et al. (2011b). In each case the model was solved in parallel on a Tesla C2075 GPU with Thrust as well as sequentially on a quad-core Intel Xeon 2.4 GHz CPU with C++. The Table also presents the sequential Matlab times reported in Judd et al. (2011b).

Table 5 shows that serial computing times increase with number of countries, \( N \), and order of polynomial approximation, \( D \), and that solving the model for many countries and high-order polynomials can be prohibitive. The regression (RLS-Tikh or RLS-TSVD) and integration (one-node monte carlo, MC(1), or monomial, M2/M1) methods also have an impact on computing times, but their effect interacts with \( N \) and \( D \) and is not uniform. For example, when \( N \) is low MC(1) typically takes much longer to compute because it involves a longer simulation (for accuracy needs). However, since the number of integration nodes in the monomial rules is a function of \( N \), their computation becomes increasingly complex as \( N \) rises. In these latter situations, the number of total operations is lower, and hence computation times faster, for MC(1) under a long simulation. Interestingly, there are a number of cases (e.g. \( N = 6 \) and \( D = 3 \)) where the GPU overturns this result: the parallelization of the integration step allows for the more complex monomial rule to be computed quite efficiently, whereas the single-node monte carlo rule does not benefit from parallelism, since it involves a single floating-point evaluation for the expectation.

The values in Table 5 also highlight other important results. First, using a compiled language such as C++ can be quite beneficial, even without parallelism: at its slowest, C++
Table 5: Timing results for the multi-country model in Judd et al. (2011b). The GPU solution was implemented with C++/Thrust using a single NVIDIA Tesla C2075 GPU; the CPU (C++) solution was implemented on a quad-core Intel Xeon 2.4 GHz CPU; the Matlab solution times were taken from Table 5 of Judd et al. (2011b). N denotes the number of countries in the model and D denotes the degree of polynomial approximation for the policy function.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$D$</th>
<th>RLS-Tikh., MC(1)</th>
<th>RLS-TSVD, M2/M1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$T=10,000, \eta = 10^{-5}$</td>
<td>$T=1000, \kappa = 10^7$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GPU</td>
<td>CPU (C++)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>82</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>224</td>
<td>238</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>580</td>
<td>594</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>73</td>
<td>78</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>316</td>
<td>330</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1905</td>
<td>1924</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>147</td>
<td>162</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1194</td>
<td>1295</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>242</td>
<td>256</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>34</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>385</td>
<td>423</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>70</td>
<td>82</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>2766</td>
<td>2781</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>490</td>
<td>528</td>
</tr>
</tbody>
</table>

provides a 2-3× speed-up over Matlab, while in many other cases it is between 10× and 40× faster. The most extreme examples are $N = 4/D = 4$ and $N = 6/D = 3$, which require roughly 10 and 18 hours, respectively, for Matlab to compute, but which are computed by C++ roughly 80× faster. This is a dramatic difference.

Finally the results demonstrate the returns to parallelism on the GPU. As mentioned above, it is not surprising that there are essentially no gains to parallelism when MC(1) is used as an integration method since the integral computation involves only a single operation which cannot be parallelized. However, when using a higher-order monomial rule, the GPU
is often $3 \times$ to $4 \times$ faster than the serial C++ implementation. When compounding the returns to using a compiled language with the returns to parallelism, there are cases where the overall computing time is $150 \times$ to $200 \times$ faster than the original Matlab software. As discussed above, the returns to parallelism could be further improved by efficiently allocating work among the GPU processors in CUDA C using thread synchronization, probably yielding a full order of magnitude speed-up over C++. Furthermore, using GSSA to solve a model that doesn’t require thread synchronization, as in the multi-country model, would probably achieve speed increases of $10 \times$ to $50 \times$ over C++ when computed on a Tesla GPU.

5 The Road Ahead

As mentioned in the introduction, the unfortunately reality is that the concepts of this paper are not timeless: developments in software and hardware will necessarily influence the way we design massively parallel algorithms. The current state of the art requires algorithmic design that favors identical execution of instructions over heterogeneous data elements, avoiding execution divergence as much as possible. Occupancy is another important consideration when parallelizing computations: most current GPUs are only fully utilized when the number of execution threads is on the order of 10,000 to 30,000. While parallelism in most algorithms can be achieved in a variety of ways, these two issues, divergence and occupancy, direct scientists to parallelize at a very fine level, with a small set of simple instructions executing on a large number of data elements. This is largely a result of physical hardware constraints of GPUs – the number of transistors dedicated to floating-point vs. memory and control-flow operation. In time, as massively parallel hardware changes, the design of algorithms suitable for the hardware will also change. And so, while this paper has provided some examples of algorithmic design for massively parallel architectures, it is most important for researchers to be aware of and sensitive to the changing characteristics of the hardware they use. The remainder of this section will highlight recent developments in parallel hardware and software and in so doing will cast our gaze to the horizon of massively parallel computing.
5.1 NVIDIA Kepler and CUDA 5

CUDA 5, the most recent toolkit released by NVIDIA on 15 October 2012, leverages the new NVIDIA Kepler architecture to increase productivity in developing GPU software. Among others, the two most notable features of CUDA 5 are dynamic parallelism and GPU callable libraries.

Dynamic parallelism is a mechanism whereby GPU threads can spawn more GPU threads directly, without interacting with a CPU. Previous to CUDA 5, all GPU threads had to be instantiated by a CPU. However, a kernel which is executed by a GPU thread can now make calls to other kernels, creating more threads for the GPU to execute. Best of all, the coordination of such threads is handled automatically by the scheduler on the GPU multiprocessor. This increases the potential for algorithmic complexity in massively parallel algorithms, as multiple levels of parallelism can be coordinated directly on the GPU. Dynamic parallelism is only available on Kepler capable NVIDIA GPUs released after 22 March 2012.

GPU callable libraries allow developers to write libraries that can be called within kernels written by other users. Prior to CUDA 5, all GPU source code had to be compiled within a single file. With the new toolkit, however, scientists can enclose GPU software in a static library that can be linked to third-party code. As high performance libraries are created, this feature will extend the capabilities of individual researchers to write application specific software, since they will be able to rely on professionally developed libraries rather than writing their own routines for each problem. An example would be simple regression or optimization routines: if an application requires such routines to called within a GPU kernel, the new CUDA toolkit allows them to be implemented in a third-party library, rather than written personally by an individual developing the particular application. GPU callable libraries only depends on CUDA 5 and not on the Kepler architecture – older NVIDIA GPUs can make use of callable libraries so long as they have the CUDA 5 drivers installed.

GPU callable libraries and dynamic parallelism interact in a way that results in a very important feature: GPU libraries that were previously only callable from a CPU can now be called directly within a kernel. As an example, CUDA BLAS, which leverages GPU parallelism for BLAS operations, can now be called by a GPU thread in order to perform...
vector or matrix operations. Prior to CUDA 5, vector and matrix operations had to be written by hand if performed within a GPU kernel. This feature, of course, will extend to other GPU libraries which spawn many threads in their implementation.

5.2 Intel Phi

On 12 November 2012, Intel released a new microprocessor known as the Intel Xeon Phi. To be specific the Phi is a coprocessor which can only be utilized in tandem with a traditional CPU that manages its operations. However, the 50 individual cores on the Phi are x86 processors in their own right, similar to x86 cores in other Intel CPU products. In other words, each Phi core possesses the capabilities of running a full operating system and any legacy software that was written for previous generation x86 CPUs.

The primary objective of the Phi is to introduce many of the advantages of massively parallel computing within an architecture that doesn’t sacrifice the benefits of traditional CPUs. At 1.05 GHz each, the 50 Phi cores don’t deliver as much raw compute power as a Tesla C2075 GPU, but they allow for far greater functionality since they have many more transistors dedicated to memory use and control flow. This effectively eliminates the issues of thread divergence and allows serial software to be more quickly and easily ported to parallel implementations.

It is difficult to forecast the nature of future parallel processors, but it is very likely that hybrid processors like the Xeon Phi will become increasingly relevant since they combine the benefits of massive parallelism with the flexibility that is necessary for a wide variety of computational tasks. Future processors may also synthesize the benefits of the Phi and current GPUs by placing heterogeneous compute cores on a single, integrated chip, overcoming memory transfer issues and simultaneously allowing for greater thread divergence within a massively parallel framework.

5.3 OpenACC

OpenACC is an example of a programming standard that allows for high-level development of parallel computation. Developed jointly by Cray, NVIDIA and PGI, OpenACC allows
users to insert compiler directives to accelerate serial C/C++ and Fortran code on parallel hardware (either a CPU or a GPU). In this way, OpenACC is very similar to OpenMP which accelerates serial code on multi-core CPUs.

OpenACC is an important example of software that promotes parallelism within software at a very high level - it requires very little effort to extend serial code to parallel hardware. With some sacrifice of efficiency and flexibility, OpenACC takes massively parallel computing into the hands of more software designers and also offers a glimpse of the future of parallel computing: software which automatically incorporates the benefits of massive parallelism with very little user interaction. Coupled with future advances in hardware, this could drastically alter the ways in which parallel algorithms are designed.

6 Conclusion

This paper has provided an introduction to current tools for massively parallel computing in economics and has demonstrated the use of these tools with examples. Sections 3 and 4 demonstrated the benefits and limitations of massively parallel computing for two specific economic problems. For example, a current NVIDIA GPU intended for scientific computing was able to speed the solution of a basic dynamic programming problem by up to 200-500×. The benefits of massive parallelism were more modest when applied to the Generalized Stochastic Simulation Algorithm (GSSA) of Judd et al. (2011b), where that particular problem (a multi-country RBC model) highlighted limitations that can arise in thread synchronization. More generally, the GSSA algorithm could attain great improvements when applied to other economic problems.

Adoption of GPU computing has been slower in economics than in other scientific fields, with the majority of software development occurring within the subfield of econometrics. Examples include Lee et al. (2010), Creel and Kristensen (2011), Durham and Geweke (2011) and Durham and Geweke (2012), all of which exploit GPUs within an MCMC or particle filtering framework. These papers demonstrate the great potential of GPUs for econometric estimation, but the examples of this paper also highlight the inherent parallelism within a much broader set of economic problems. The truth is that almost all computationally
intensive economic problems can benefit from massive parallelism – the challenge is creatively finding the inherent parallelism, a task which often involves changing the way the problem is traditionally viewed or computed.

The intent of the examples in this paper is to demonstrate how traditional algorithms in economics can be altered to exploit parallel resources. This type of thought process can then be applied to other algorithms. However, since the tools of massive parallelism are ever changing, so will the design of parallel algorithms. The current architecture of GPUs guides the development of parallel software since it places limitations on memory access and control flow, but as these aspects are likely to change with the development of new many-core and heterogeneous processors, the ability to perform parallel computations on many data elements will also change. The overriding objective then is to creatively adapt algorithms for new and changing architectures.

As time progresses, parallel computing tools are becoming more accessible for a larger audience. So why learn the nuts and bolts of massively parallel computing now? Why not wait a couple of years until it is even more accessible? For many researchers, waiting might be the optimal path. However, a frontier will always exist and pushing the frontier will not only yield returns for computationally challenging problems, but it will also inform economists’ choices about paths for future research. For the economist that is tackling computationally intensive problems and that is often waiting long periods of time for a computer to yield solutions, becoming fluent in the tools of this paper and staying at the frontier will pay great dividends.

References


Figure 3: Second-order polynomials $ax^2 + 2.3x + 5.4$ for $a \in [-0.9, -0.1]$. The darkest line corresponds to $a = -0.1$. 