MASSIVELY PARALLEL COMPUTATION USING GRAPHICS PROCESSORS WITH APPLICATION TO OPTIMAL EXPERIMENTATION IN DYNAMIC CONTROL

SERGEI MOROZOV AND SUDHANSHU MATHUR

Abstract. The rapid increase in the performance of graphics hardware, coupled with recent improvements in its programmability has lead to its adoption in many non-graphics applications, including wide variety of scientific computing fields. At the same time, a number of important dynamic optimal policy problems in economics are athirst of computing power to help overcome dual curses of complexity and dimensionality. We investigate if computational economics may benefit from new tools on a case study of imperfect information dynamic programming problem with learning and experimentation trade-off, that is, a choice between controlling the policy target and learning system parameters. Specifically, we use a model of active learning and control of linear autoregression with unknown slope that appeared in a variety of macroeconomic policy and other contexts. The endogeneity of posterior beliefs makes the problem difficult in that the value function need not be convex and policy function need not be continuous. This complication makes the problem a suitable target for massively-parallel computation using graphics processors (GPUs). Our findings are cautiously optimistic in that the new tools let us easily achieve a factor of 15 performance gain relative to an implementation targeting single-core processors. Further gains up to a factor of 26 are also achievable but lie behind a learning and experimentation barrier of its own. Drawing upon experience with CUDA programming architecture and GPUs provides general lessons on how to best exploit future trends in parallel computation in economics.

Keywords: Graphics Processing Units · CUDA programming · Dynamic programming · Learning · Experimentation

JEL Classification: C630 · C800

1. Introduction

Under pressure to satisfy insatiable demand for high-definition real-time 3D graphics rendering in the PC gaming market, Graphics Processing Units (GPUs) have evolved over the past decade far beyond simple video graphics adapters. Modern GPUs are not single processors but rather are programmable, highly parallel multi-core computing engines with supercomputer-level high performance floating point capability and memory bandwidth. They commonly reach speeds of hundreds of billions of floating point operations per second (GFLOPS) and some contain over a billion transistors.

Because GPU technology benefits from large economies of scale in the gaming market, such supercomputers on a plug-in board have become very inexpensive for the raw horsepower they provide. Scientific community realized that this capability could be put to use for general purpose computing. Indeed, many mathematical computations, such as matrix multiplications or random number generation, which are required for complex visual and physics simulations in games are also the same computations prevalent in a wide variety of scientific computing applications from computational fluid dynamics to signal processing to cryptography to computational biochemistry. Graphics card manufacturers, such as AMD/ATI and NVIDIA, has supported the trend toward the general purpose computing by widening the performance edge, by making GPUs more programmable, by

Date: September 6, 2010.

Version 1.09.

We thank participants of 2009 International Conference on Computing in Economics and Finance and 2010 Econometric Society World Congress for insightful comments. All remaining errors are our own.

including additional functionality such as single and double precision floating point capability and by releasing software development kits.

The advent of GPUs as a viable tool for general purpose computing parallels the recent shift in the microprocessor industry from maximizing single-core performance to integrating multiple cores to distributed computing (Creel and Goffe, 2008). GPUs are remarkable in the level of multi-core integration. For example, high-performance enthusiast GeForce GTX280 GPU from NVIDIA contains 30 multiprocessors each consisting of eight scalar processor cores, for a total of 240 (NVIDIA, 2008). As each scalar processor core is further capable of running multiple threads, it is clear that GPUs represent the level of concurrency today that cannot be found in any other consumer platform. Inevitably, as CPU-based computing is moving in the same massively multi-core ("manycore") direction, it is time now to re-think the algorithms to be aggressively parallel. Otherwise, if the solution is not fast enough, it will never be (Buck, 2005).

Parallel computing in economics is not widespread but does have fairly long tradition. Chong and Hendry (1986) developed an early parallel Monte Carlo simulation for econometric evaluation of linear macro-economic models. Coleman (1992) takes advantage of parallel computing to solve discrete-time nonlinear dynamic models expressed as recursive systems with an endogenous state variable. Nagurney, Takayama, and Zhang (1995) and Nagurney and Zhang (1998) use massively parallel supercomputers to model dynamic systems in spatial price equilibrium problems and traffic problems. Doornik, Hendry, and Shephard (2002) provide simulation-based inference in a stochastic volatility model, Ferrall (2003) optimizes finite mixture models in parallel, while Swann (2002) develops parallel implementation of maximum likelihood estimation. A variety of econometric applications for parallel computation is discussed in Doornik, Shephard, and Hendry (2006). Sims, Waggoner, and Zha (2008) employ grid computing tools to study Bayesian algorithms for inference in large-scale multiple equation Markov-switching models. Tutorial of Creel and Goffe (2008) urges further application of parallel techniques by economists, whereas Creel (2005) identifies steep learning curve and expensive hardware as the main adoption barriers. None of these studies take advantage of GPU technology.

Financial engineering turned to parallel computing with the emergence of complex derivative pricing models and popular use of Monte-Carlo simulations. Zenios (1999) offers an early synthesis of the developments of high-performance computing in finance. Later work includes Pflug and Swietanowski (2000) on parallel optimization methods for financial planning under uncertainty, Abdelkhalek, Bilas, and Michaelides (2001) on parallelization of portfolio choice, Rahman, Thulasiram, and Thulasiraman (2002) on neural network forecasting stock prices using cluster technology, Kola, Chhabra, Thulasiram, and Thulasiraman (2006) on real-time option valuation, etc. Perhaps due to better funding and more acute needs, quantitative analysts on Wall Street trading desks took note of the GPU technology (Bennemann, Beinker, Eggloff, and Guckler, 2008) ahead of academic economists.

To the best of our knowledge, ours is the first attempt to accelerate economic decision-making computations on graphics chips. Our application of choice is from a difficult class of imperfect information dynamic programs. In this class, there is an information exploration-exploitation tradeoff, that is, a choice between learning system parameters and controlling the policy target. Specifically, we use a model of active learning and control of linear autoregression with unknown slope. The model is a version of Beck and Wieland's (2002) Bayesian dual control model that shuts down parameter drift.[1] As the system evolves, new data on both sides of the state evolution equation force revisions of estimates for location and precision parameters that characterize posterior beliefs. These evolving beliefs thereby become part of the three-dimensional system state vector to keep track of. The dimension of the state vector matters not only because Bayes rule is nonlinear but, more importantly, because the value function need not be convex, and policy function need not be continuous (Kendrick, 1978; Easley and Kiefer, 1988; Wieland, 2000). Functional approximation methods that rely on smoothness therefore do not work and one is compelled to use brute-force

[1]Without constant term, our model also could be obtained as a special case of the one studied in Morozov (2008) by imposing known persistence.

discretization. Endogeneity of information is what makes the problem so complex even when state dimension is low. This difficulty makes the problem a suitable target for GPU-based computation.

The most compute-intensive part of the algorithm is a loop updating value function at all points on three-dimensional grid. Since these points can be updated independently, the problem can be parallelized easily. For multi-core CPU-based computation we use OpenMP compiler directives (Chandra, Menon, Dagum, and Kohr, 2000; Chapman, Jost, and van der Paas, 2007) to generate as many as four treads to run on a state-of-the-art workstation with a quad-core CPU. For GPU-based computation, we use NVIDIA's CUDA platform in conjunction with GeForce GTX280 card supporting this technology. The promise of GPU acceleration is realized as we see initial speedups up to a factor of 15 relative to optimized CPU implementation. With some additional performance tuning we are able to reach a factor of 26 in some cases.

The paper is laid out as follows. Section 2 explains the new concepts of GPU programming and available hardware and software resources. Sections 3 and 4 are dedicated to our case study of the imperfect information dynamic programming problem. The former sets up theoretical background, while the latter contrasts CPU- and GPU-based approaches in terms of program design and performance. Section 5 summarizes our findings and offers a vision of what's to come.

2. GPU Programming

2.1. **History.** Early attempts to harness GPUs for general purpose computing (so called GPGPU) had to express their algorithms using existing graphics application programming interfaces (APIs): OpenGL (Kessenich, Baldwin, and Rost, 2006) and DirectX (Bargen and Donnelly, 1998). The approach was awkward and thus unpopular.

Over the past decade, graphics cards evolved to become programmable GPUs and on to become fully programmable data-parallel floating-point computing engines. The two largest discrete GPU vendors, AMD/ATI and NVIDIA, have supported this trend by releasing software tools to simplify the development of GPGPU applications and adding hardware support to use the increasingly parallel GPU hardware without the need for computations to be cast as graphics operations. In 2006, AMD/ATI released Close-to-the-metal (CTM), a relatively low-level interface for GPU programming that bypasses the graphics API. NVIDIA, also in 2006, took a more high-level approach with its Compute Unified Device Architecture (CUDA) interface library. CUDA extends C programming language to allow the programmer to delegate portions of the code for execution on GPU.[2] Although similar AMD FireStream technology later included software development kit with higher-level language support, NVIDIA CUDA ecosystem was the most mature at the inception of our project and was selected as a testbed. Further developments aimed at leveraging many-core coprocessors include OpenCL cross-platform effort (Khronos OpenCL Working Group, 2009), Intel Ct programming model (Ghuloum, Sprangle, Fang, Wu, and Zhou, 2007), and Microsoft DirectCompute (Boyd and Schmit, 2009).[3]

General purpose computing on GPUs has come a long way since the initial interest in the technology in 2003-2004. On the software side, there are now major applications across the entire spectrum of high performance computing, except perhaps computational economics. On the hardware side, many of the original limitations of GPU architectures have been removed and what remains is mostly related to the inherent trade-offs of massively threaded processors.

2.2. **Data Parallel Computing.** While there are already tools available that enable parallel processing, these tools are largely dedicated to task parallel models. The task parallel model is built around the idea that parallelism can be extracted by constructing threads that each have their own goal or task to complete. While most parallel programming is task parallel, there is another form of parallelism that can be exploited.

In contrast to the task parallel model, data parallel programming runs the same block of code on hundreds or even thousands of data points. Whereas typical multi-threaded program to be executed

[2]Third party wrappers are also available for Python, Fortran, Java and Matlab.

[3]Heterogeneous cross-vendor device management can be very tedious. See Kirk and Hwu (2010) for some illustrative examples.

on moderately multi-core/multi-CPU computer handles only small number of threads (typically no more than 32), a data parallel program doing something like image processing may spawn millions of threads to do the processing on each pixel. The way these threads are actually grouped and handled will depend on both the way the program is written and the hardware the program is running on. CUDA is one way to implement data parallel programming.

2.3. **CUDA Programming.** As already mentioned, CUDA is a general purpose data-parallel computing interface library. It consists of runtime library, set of function libraries, C/C++ development toolkit, extensions to the standard C programming language and a hardware abstraction mechanism that hides GPU hardware from developers. It allows heterogeneous computation mixing conventional code targeting host CPU with data parallel code for GPU acceleration. Like OpenMP (Chandra, Menon, Dagum, and Kohr, 2000) and unlike MPI (Gropp, Lusk, Skjellum, and Thakur, 1999), CUDA adheres to the shared memory model. Furthermore, although CUDA requires writing special code for parallel processing, explicitly managing threads is not required.

CUDA hardware abstraction mechanism exposes a virtual machine consisting of a large number of *streaming multi-processors* (SMs). A multiprocessor consists of 8 scalar processors (SPs), each capable of executing independent threads. Each multiprocessor has four types of on-chip memory: one set of registers per SP, shared memory, constant read-only memory cache and read-only texture cache.

The main programming concept in CUDA is the notion of a *kernel* function. A kernel function is a single subroutine that is invoked simultaneously across multiple thread instances. The threads are organized into one-, two-, or three-dimensional blocks which in turn are laid out on a two-dimensional grid. The blocks are completely independent of each other, whereas threads within a block are mapped entirely and execute to completion on a single streaming multiprocessor. This allows memory sharing and synchronization using on-chip memory. In order to optimize a CUDA application, one should try to achieve an optimal balance between the size and number of blocks. More threads in a block reduce the effect of memory latencies, but it will also reduce the number of available registers.

Every block is comprised of several groups of 32 threads called *warps*. All threads in the same warp execute the same program. Execution is the most efficient if all threads in a warp execute in lockstep. Otherwise, threads in a warp diverge, i.e. follow different execution paths. If this occurs, the different execution paths have to be serialized causing performance loss.

CUDA's extensions to the C programming language are relatively minor. Each function declaration can include a *function type qualifier* determining whether the function will execute on the CPU or GPU and if it is a GPU function, whether it is callable from the CPU. Variable declarations also include qualifiers specifying where in the memory hierarchy the variable will reside. Finally, kernels have special thread-identification variables while calls to GPU functions must include an *execution configuration* specifying grid and thread-block configuration and allocations of shared memory on each SM. Functions executed by a GPU may have certain limitations depending on hardware and software revisions. For example, no function recursion, no static variables inside functions or variable number of arguments may be allowed. Two memory management types are supported: linear memory with access by 32-bit pointers, and CUDA-arrays with access only through texture fetch functions.

Managing memory hierarchy is another key to high performance. Since there are several kinds of memory available on GPUs with different access times and sizes, the effective bandwidth can vary significantly depending on the access pattern for each type of memory, ordering of the data access, use of buffering to minimize data exchange between CPU and GPU, overlapping inter-GPU communication with computation, etc (Kirk and Hwu, 2010; NVIDIA, 2008). Memory access to local shared memory including constant and texture memory and well aligned access to global memory are particularly fast. Indeed, ensuring proper memory access can achieve a large fraction of the theoretical peak memory bandwidth which is in the order of 100 GB/sec for today's GPU boards.

The main CUDA process works on a host CPU from where it initializes a GPU, distributes video and system memory, copies constants into video memory, starts several copies of kernel processes on

a graphics card, copies results from video memory, frees memory, and shuts down. CUDA-enabled programs can interact with graphics APIs, for example to render data generated in a program. Multiple GPUs can be used simultaneously for further speed gains (Sanders and Kandrot, 2010).

Files of the source CUDA C or C++ code are compiled with `nvcc`, which is just a shell to other tools: `cudacc`, `g++`, `cl`, etc. `nvcc` generates: CPU code, which is compiled together with other parts of the application, written in pure C or C++, and PTX object code for GPU.

NVIDIA CUDA architecture for GPU computing is a good solution for a wide circle of parallel tasks. It works with many NVIDIA processors. It improves the GPU programming model, making it much simpler and adding a lot of features, such as shared memory, thread synchronization, double precision, and integer operations. CUDA technology is available freely to every software developer, any C programmer. It is the learning curve that is the steepest adoption barrier. The NVIDIA CUDA technology is now being taught at universities around the world, typically in computer science or engineering departments. New textbooks and self-study guides are being written, e.g. Kirk and Hwu (2010); Sanders and Kandrot (2010). Numerous seminars, tutorials, technical training courses and third-party consulting services are also available to help one get started. Website http://www.nvidia.com/object/cuda_education.html collects some of these resources.

CUDA has some disadvantages. This architecture works only with recent NVIDIA's GPUs – starting from GeForce 8 and 9, and Quadro/Tesla. Writing code is not automatic. In section 4 we'll show how much progress we made in the quest for performance and whether it was difficult to do.

3. Case Study: Dynamic programming solution of learning and active experimentation problem

In this section we introduce the imperfect information optimal control problem, dynamic programming approach as well as a couple of useful suboptimal policies.

3.1. Problem Formulation.

The decision-maker minimizes discounted intertemporal cost-to-go function with quadratic per-period losses,

$$\min_{\{u_t\}_{t=0}^{\infty}} \mathbb{E}_0 \left[\sum_{t=0}^{\infty} \delta^t \left((x_t - \bar{x})^2 + \omega (u_t - \bar{u})^2 \right) \right], \tag{1}$$

subject to the evolution law of policy target x_t from a class of linear first-order autoregressive stochastic processes

$$x_t = \alpha + \beta u_t + \gamma x_{t-1} + \epsilon_t, \qquad \epsilon_t \sim \mathcal{N}(0, \sigma_\epsilon^2). \tag{2}$$

$\delta \in [0, 1)$ is the discount factor, $\bar{x}$ is the stabilization target, $\bar{u}$ is the "costless" control, $\omega \geq 0$ gives weight to the deviations of u_t from $\bar{u}$.[4] Variance of the shock, σ_ϵ^2 is known, and so are the constant term α and autoregressive persistence parameter $\gamma \in (-1, 1)$. Equations (1)-(2) are a stylized representation of many macroeconomic policy problems, such as monetary or fiscal stabilization, exchange rate targeting, pricing of government debt, etc.

Only one of the parameters that govern the conditional mean of x_t, namely the slope coefficient β is unknown. Initially, prior belief about β is Gaussian:

$$\beta \sim \mathcal{N}(\mu_0, \Sigma_0). \tag{3}$$

Gaussian prior (3) combined with normal likelihood (2) yields Gaussian posterior (Judge, Lee, and Hill, 1988), and so at each point in time the belief about unknown β is conditionally normal and is completely characterized by mean μ_t and variance Σ_t (sufficient statistics). Upon observing a

[4]Under monetary policy interpretation of the model, ω describes flexibility of monetary policy with respect to its dual objectives of inflation and output stability (Svensson, 1997).

realization of x_t, these are updated in accordance with the Bayes law:[5]

$$\begin{aligned}\Sigma_{t+1} &= \left(\Sigma_t^{-1} + \frac{1}{\sigma_\epsilon^2}u_t^2\right)^{-1},\\ \mu_{t+1} &= \Sigma_{t+1}\left(\frac{1}{\sigma_\epsilon^2}u_t x_t + \Sigma_t^{-1}\mu_t\right).\end{aligned} \tag{4}$$

Note that the evolution of the variance is completely deterministic.

Under distributional assumptions (2) and (3), the imperfect information problem is transformed into a state-space form by defining the extended state containing both physical and informational components:

$$S_t = (x_t, \mu_{t+1}, \Sigma_{t+1})' \in \mathcal{S} \subseteq \mathbb{R}^3. \tag{5}$$

As a useful shorthand, we encode policy target process (2) and Bayesian updating (4) via mapping

$$S_{t+1} = B(S_t, x_{t+1}, u_{t+1}). \tag{6}$$

3.2. **Dynamic programming.** The objective is to find optimal policy function $u^*(S)$ that minimizes intertemporal cost-to-go (1) subject to the evolution of extended state (6), given an initial state S_0. It is also of interest to compare the value (i.e. cost-to-go) of the optimal policy to those of certain simple alternative policy rules. Thus, we will also require computation of cost-to-go functions of some simple policies.

3.2.1. *Inert uninformative policy.* So called inert uninformative policy simply sets control impulse to zero, regardless of current physical state x_t or current beliefs. Such policy is not informative for Bayesian learning in that it leaves posterior beliefs exactly equal to the prior beliefs. In turn, this allows closed-form solution for the corresponding cost-to-go function:

$$\begin{aligned}V^0(S_t) = &\frac{(\alpha+\gamma x_t-\bar{x})^2 - \delta\gamma\left((\bar{x})^2-\alpha^2-\gamma\bar{x}(2\alpha-\bar{x})+\gamma x_t^2(1+\gamma)-2x_t(\bar{x}-\alpha+\gamma^2\bar{x})\right)}{(1-\delta)(1-\gamma\delta)(1-\gamma^2\delta)}\\ &+\frac{\gamma^3\delta^2\left(x_t-\bar{x}\right)^2}{(1-\delta)(1-\gamma\delta)(1-\gamma^2\delta)}+\frac{\sigma_\epsilon^2}{(1-\delta)(1-\gamma^2\delta)}+\frac{\omega\left(\bar{u}\right)^2}{1-\delta}.\end{aligned} \tag{7}$$

Omitting time subscripts, it could be shown that the inert policy cost-to-go function V^0 satisfies a recursive relationship:

$$V^0(x,\mu,\Sigma) = (\alpha+\gamma x-\bar{x})^2 + \omega\bar{u}^2 + \sigma_\epsilon^2 + \delta\mathbb{E}V^0(\alpha+\gamma x+\epsilon,\mu,\Sigma). \tag{8}$$

Relationship (8) can serve as a basis of an iterative computational algorithm, starting from any simple initial guess, for example $\widetilde{V}^0 \equiv 0$. Upon convergence, the recursive algorithm, policy iteration in disguise, should approximately recover (7). This provides simple test of correctness of our CPU-based and GPU-based computations.[6]

Another use of the inert uninformative policy is to provide explicit bounds on the optimal policy with experimentation u_{t+1}^* given current state S_t via simple quadratic inequality

$$\mathbb{E}_t\left[(x_{t+1}-\bar{x})^2+\omega\left(u_{t+1}^*-\bar{u}\right)^2\right] \le \mathbb{E}_t\left[\sum_{\tau=1}^{\infty}\delta^{\tau-1}\left((x_{t+\tau}-\bar{x})^2+\omega(u_{t+\tau}^*-\bar{u})^2\right)\right] \le V^0\left(S_t\right). \tag{9}$$

Asymptotically, the bounds are linear in the x direction, converge to a positive constant in the μ direction and converge to zero in the Σ direction.

[5]We use subscript $t+1$ to denote beliefs after x_t is realized but before the choice of u_{t+1} is made at the beginning of period $t+1$. This notational timing convention accords with that in Wieland (2000). Technically, it means that u_{t+1} is measurable with respect to filtration $\mathcal{F}_t$ generated by histories of stochastic process up until time t.

[6]Since our numerical implementation restricts the state space to a three-dimensional hypercube and assumes constant cost-to-go function beyond its boundary, analytic and numerical solutions will necessarily be different near that boundary. However, if the numerical solution is implemented correctly, its final values at any fixed interior point will converge to the analytical solution as hyper-cube is progressively expanded while the grid spacing remains constant. This is indeed what we have observed.

3.2.2. *Cautionary myopic policy.* Cautionary myopic policy takes account of coefficient uncertainty but disregards losses incurred in periods beyond current. It optimizes the expected one-period-ahead loss function

$$
\begin{aligned}
L(S_t, u_{t+1}) &= \int \left((\alpha + \beta u_{t+1} + \gamma x_t + \epsilon_{t+1} - \bar{x})^2 + \omega (u_{t+1} - \bar{u})^2 \right) p(\beta|S_t) q(\epsilon_{t+1}) d\beta d\epsilon_{t+1} \\
&= \left(\Sigma_{t+1} + \mu_{t+1}^2 + \omega \right) u_{t+1}^2 + 2 \left((\mu_{t+1}\gamma) x_t - \mu_{t+1}\bar{x} - \omega \bar{u} \right) u_{t+1} \\
&+ \omega \bar{u}^2 + \gamma^2 x_t^2 + \bar{x}^2 + \sigma_\epsilon^2 - 2\gamma \bar{x} x_t,
\end{aligned} \tag{10}
$$

where $p(\beta|S_t)$, $q(\epsilon_t)$ represent posterior belief density and density of state shocks, respectively. The solution can be found in closed form:

$$
u_{t+1}^{MYOP} = -\frac{\mu\gamma}{\Sigma + \mu^2 + \omega} x_t + \frac{\mu(\bar{x} - \alpha) + \omega\bar{u}}{\Sigma + \mu^2 + \omega}. \tag{11}
$$

Cautionary myopic policy is a useful and popular benchmark in studies of value of experimentation (Prescott, 1972; Easley and Kiefer, 1988; Lindoff and Holst, 1997; Wieland, 2000; Brezzia and Lai, 2002). From (9), it follows that myopic policy rule is precisely the mid-point of the explicit bounds on the optimal policy with experimentation. Thus, it is likely to be a good initial guess for the optimization algorithm searching for actively optimal policy with experimentation, see sections 3.2.3 and 4.1.

Cost-to-go function of the cautionary myopic policy is not explicit but satisfies recursive functional equation analogous to (8):

$$
\begin{aligned}
V^{MYOP}(x, \mu, \Sigma) &= \mathbb{E} \left(\alpha + \beta u^{MYOP}(x, \mu, \Sigma) + \gamma x - \bar{x} \right)^2 + \omega (u^{MYOP}(x, \mu, \Sigma) - \bar{u})^2 + \sigma_\epsilon^2 \\
&+ \delta \mathbb{E} V^{MYOP} \left(\alpha + \beta u^{MYOP}(x, \mu, \Sigma) + \gamma x + \epsilon, \mu', \Sigma' \right),
\end{aligned} \tag{12}
$$

where μ' and Σ' are future beliefs given by Bayes updating (4).[7] A method to find approximate solution of (12) is policy iteration on a discretized state space (i.e. on a grid). The iteration starts with an arbitrary initial guess which is then plugged into the right hand side of (12). Improved guess is obtained upon evaluation of the left hand side at every gridpoint. The process is continued until convergence to an approximate cost-to-go function of the cautionary myopic policy. For a formal justification of the algorithm, see Bertsekas (2005, 2001).

3.2.3. *Optimal policy with experimentation.* Unlike the two previous policies, the optimal policy that take full account of the value of experimentation is not explicit. It is given by a solution of the Bellman functional equation of dynamic programming. It is given by

$$
V(S_t) = \min_{\{u_{t+1}\}} \left\{ L(S_t, u_{t+1}) + \delta \int V(S_{t+1}) \, p(\beta, \epsilon_{t+1}|S_{t+1}) d\beta d\epsilon_{t+1} \right\}, \tag{13}
$$

where $L(S_t, u_{t+1})$ as in (10). Although the stochastic process under control is linear and the loss function is quadratic, the belief updating equations are non-linear, and hence the dynamic optimization problem is more difficult than those in the class of linear quadratic problems. Following Easley and Kiefer (1988), it could be shown that Bellman functional operator is a contraction and a stationary optimal policy exists such that corresponding value function is continuous and satisfies the above Bellman equation. Solution of (13) is a mapping $u^* : \mathcal{S} \to \mathbb{R}$ from extended state space to policy choices. Based on above theoretical arguments, an approximate solution can be obtained by recursive use of discretized version of (13) starting from some initial guess (i.e. value iteration). Upon convergence, one is left with both approximate policy and cost-to-go functions. This process is computationally more demanding than for earlier two policies due to an additional minimization step.

[7] It is this nonlinear dynamics of beliefs that precludes closed-form computation of the value function. Under constant beliefs, the cost-to-go function would be quadratic in x, with explicit closed-form coefficients.

4. CPU-based versus GPU-based Computations

4.1. **CPU-based computation.** The approximations to optimal policy and value function calculations follow the general recursive numerical dynamic programming methods outlined above. Purely for simplicity, we omit several acceleration techniques. In particular, we do not introduce alternating approximate policy evaluation steps and asynchronous Gauss-Seidel-type sweeps of the state space (Morozov, 2008, 2009a,b). Those are beneficial but camouflage the benefits accruing to massively multithreaded implementation.

Since the integration step in (13) (as well as integration step implicit in (8) and (12)) cannot be carried out analytically, we resort to the Gauss-Hermite quadrature. Further, the actively optimal policy and cost-to-go functions are represented by means of multi-linear interpolation on the non-uniform tensor (Kroneker) product grid in the state space. The non-uniform grid is designed to place grid-points more densely in the areas of high curvature, namely in the vicinity of $x = \bar{x}$ and $\mu = 0$. The grid is uniform along Σ dimension. Although, in principle, the state space is unbounded, we restricted our attention to the three-dimensional hyper-cube. The boundaries were chosen via a priori simulation experiment to ensure that high curvature regions are completely covered and that all simulated sequences originating sufficiently deep inside the hyper-cube remain there for the entire time span of a simulation. The dynamic programming algorithm was iterated to convergence with relative tolerance[8] of $1e-6$ for the two suboptimal policies and $1e-4$ for the optimal policy. Univariate minimization uses safeguarded multiple restart version of Brent's golden section search (Brent, 1973).[9]

Table 1 reports runtimes and memory usage of CPU-based computations for the three types of policies. These were implemented in Fortran90 with outermost loop over the state space explicitly parallelized using OpenMP directives (Chandra, Menon, Dagum, and Kohr, 2000; Chapman, Jost, and van der Paas, 2007). The code was compiled in double precision[10] with Intel Fortran compiler for 64-bit systems, version 11.0,[11] and run on a high performance quad-core single CPU workstation utilizing Core i7 Extreme Edition 965 Nehalem CPU overclocked to 3.4 GHz. In order to reduce timing noise, the codes were run four times and compute times averaged out for each policy type, CPU thread count and gridsize. It should be clear that the deck is intentionally stacked in CPU favor.

Table 1 serves to emphasize very good scaling of numerical dynamic programming with the thread count, since communication among different threads is not required and workload per thread is fairly uniform. For small grid sizes overhead of thread creation and destruction dominates performance benefit of multiple threads.[12] At large grid sizes, memory becomes limiting factor both in terms of ability to store cost-to-go function and to quickly update it in memory.

4.2. **GPU-based computation.** For our GPU-based computations we used graphics card featuring GTX280 chip. It has 1.4 billion transistors, theoretical peak performance of 933 Gflops in single

[8]Relative tolerance was defined as relative sup norm distance between consecutive approximations magnified by a factor of $\delta/(1-\delta)$ based on MacQueen-Porteus bounds (Porteus and Totten, 1978; Bertsekas, 2001).

[9]Preliminary efforts to ensure robustness of minimization step involved testing our method against Nelder-Meade simplex method (Spendley, Hext, and Himsworth, 1962; Nelder and Mead, 1965; Lagarias, Reeds, Wright, and Wright, 1998) with overdispersed multiple starting points (random or deterministic), simulated annealing (Kirkpatrick, Gelatt Jr., and Vecchi, 1983; Goffe, Ferrier, and Rogers, 1994), direct search (Conn, Gould, and Toint, 1997; Lewis and Torczon, 2002), genetic algorithm (Goldberg, 1989) and their hybrids (Zabinsky, 2005; Horst and Paradalos, 1994; Paradalos and Romeijn, 2002). They all yielded virtually identical results, with occasional small random noise due to stochastic nature of the search for optimum. On the other hand, these alternative methods, so well suited to nonconvex and nonsmooth problems, required significant computational expense, driven primarily by the sharp increase in the number of function evaluations. Our choice is a compromise between robustness and speed.

[10]Same code compiled in single precision required more dynamic programming iterations to achieve convergence to the same tolerance, outweighing speed benefits of lower precision.

[11]Intel compilers (Fortran and C) performed significantly better then those from GNU complier collection (`gcc` and `gfortran`).

[12]More elaborate performance-oriented implementation would vary number of threads depending on the size of the problem in order to avoid performance degradation due to thread creation.

TABLE 1. Performance scaling of CPU-based computation under different policies.

Gridsize	**Grid points**	**CPU Threads**	**Inert Uninformative**		**Myopic**		**Optimal**	
			CPU Time	Memory Usage	CPU Time	Memory Usage	CPU Time	Memory Usage
8x8x8	512	1	9.96E-002	15M	9.36E-002	15M	1.43	16M
		2	0.16	19M	0.15	19M	0.74	20M
		4	0.27	28M	0.22	93M	0.57	94M
16x16x16	4,096	1	0.95	15M	1.01	15M	13.22	16M
		2	0.47	19M	0.52	19M	7.03	86M
		4	0.33	28M	0.35	94M	4.79	94M
32x32x32	32,768	1	8.42	16M	9.72	16M	122.68	18M
		2	4.37	20M	4.97	20M	63.55	87M
		4	2.71	94M	3.01	94M	37.56	96M
64x64x64	262,144	1	77.17	24M	94.07	21M	1,085.47	29M
		2	39.45	28M	47.5	91M	559.10	98M
		4	21.73	99M	26.14	99M	344.43	103M
128x128x128	2,097,152	1	798.45	81M	962.46	64M	9,972.39	111M
		2	392.26	85M	491.41	68M	5,300.80	179M
		4	211.18	93M	270.37	138M	3,131.72	187M
256x256x256	16,777,216	1	7,368.56	526M	9,880.06	398M	98,809.89	783M
		2	3,759.02	530M	5,159.7	402M	51,161.30	787M
		4	2,016.57	602M	2,855.14	474M	29,972.30	860M

precision, peak memory bandwidth of 141.7 GB/sec and capability to work on 30,720 threads simultaneously.[13] For the software stack, we used CUDA version 2.3 for Linux, including associated runtime, compiler, debugger, profiler and utility libraries.

GPU-based computation mirrors CPU-based one in all respects except for how it distributes the work across multiple threads. Code fragments below illustrate the differences in the implementation of the main sweep over all points on the grid between the two programming frameworks. To focus on the key details, we selected these fragments from a codebase for the evaluation of the cost-to-go function of the cautionary myopic policy. The code for the optimal policy is conceptually similar but is obscured by the details of implementing the minimization operator. We also omit parts of the code that are not interesting, such as checking iteration progress, reading input or generating output files. Omitted segments are marked by ellipses.

Fortran 90 code

```
1  ...
2  ! set multithreaded OpenMP version using all available CPUs
3  #ifdef _OPENMP
4    call OMP_SET_NUM_THREADS(numthreads)
5  #endif
6  allocate(V(NX,Nmu,NS,2),U(NX,Nmu,NS))
7  ...
8  supval = abs(10.0d0*PolTol+1.0)
9  polit1=0
10 ip = 1
11 ppass =0
12 do while((ip<MaxPolIter+1).and.(ppass.eq.0))
13  ! loop over the grid of the three state variables
14  !$omp parallel default(none) &
15  !$omp shared(NX,Nmu,NS,U,V,X,mu,Sigma,alpha,gamma,delta,omega,ubar,xbar,sigma_eps) &
16  !$omp private(i,j,k)
17  !$omp do
18  do i=1,NX
19     do j=1,Nmu
20        do k=1,NS
21           V(i,j,k,2) = F(U(i,j,k),i,j,k,V)
22        enddo
23     enddo
24  enddo
25  !$omp end do
26  !$omp end parallel
27
28  ! check convergence criterion for the policy function iteration
29  supval = maxval(abs(V(:,:,:,1)-V(:,:,:,2)))
30  checkp = maxval(delta*abs(V(:,:,:,1)-V(:,:,:,2))/((1-delta)*(abs(V(:,:,:,1)))))
31  if (checkp<PolTol) then
32     ppass = 1
33  endif
34
35  ! update the value function
36  V(:,:,:,1) = V(:,:,:,2)
37
38  print *, '        FINISHED POLICY ITERATION',ip
39  if (ppass.eq.1) then
40     print *, 'POLICY ITERATION CONVERGED'
41  else
42     print *, 'POLICY ITERATIONS - NO CONVERGENCE'
43  endif
44  ip=ip+1
45 enddo
46 polit1=ip-1
47 ...
```

The Fortran code is fairly straightforward. The main loop starts on line 14 and encloses pointwise value updates. So called OpenMP *sentinels* on lines 14-17 and 25-26 tell compiler to generate multiple threads that will execute the loop in parallel. Division of the workload is up to compiler to decide.

Cuda code (main)

```
1  ...
2    cudaMalloc((void**) &dX,NX*sizeof(double));
```

[13] Standard configuration is clocked at 602 MHz for the core, 1296 MHz for shader units, and 2214 MHz for the memory. Higher performing versions are already available.

```
3  ...
4    cudaMemcpy(dX,X,NX*sizeof(double),cudaMemcpyHostToDevice);
5  ...
6    numBlocks=512;
7    numThreadsPerBlock=64;
8    dim3 dimGrid(numBlocks);
9    dim3 dimBlock(numThreadsPerBlock);
10
11   // start value iteration cycles
12   supval=fabs(10.0*PolTol+1.0);
13   polit1=0;
14   ip=1;
15   ppass=0;
16   while ((ip<MaxPolIter+1)&&(ppass==0))
17   {
18     // update expected cost-to-go function on the whole grid (in parallel)
19     UpdateExpectedCTG_kernel<<<dimGrid,dimBlock>>>(dU,dX,dmu,dSigma,dV0,drno,dwei,dV1);
20     cudaThreadSynchronize();
21
22     // move the data from device to host to do convergence checks
23     cudaMemcpy(V1,dV1,NX*Nmu*NS*sizeof(double),cudaMemcpyDeviceToHost);
24
25     // check convergence criteria for the policy function iteration if ip>=2
26 ...
27     // update value function, directly on the device
28     cudaMemcpy(dV0,dV1,NX*Nmu*NS*sizeof(double),cudaMemcpyDeviceToDevice);
29     // update value function on host as well
30     cudaMemcpy(V0,V1,NX*Nmu*NS*sizeof(double),cudaMemcpyHostToHost);
31 ...
32   }
33 ...
34   cudaFree(dX);
35 ...
```

CUDA code requires separate memory spaces allocated on the host CPU and on the graphics device, with explicit transfers between the two. It replaces entire loop with a call to a *kernel* function (line 19). Kernel function is executed by each thread applying device function to its own chunk of data. The number of threads and their organization into blocks of threads is an important tuning parameter. More threads in a block hide memory latencies better but also reduce resources available to each thread. For a rough guidance on the tradeoff between the size and the number of blocks we used NVIDIA's CUDA occupancy calculator, a handy spreadsheet tool. Even though NVIDIA recommends number of threads per block that is a multiple of available streaming multiprocessors (30 for GTX280), we were restricted to the powers of two by our implemntation of divide-and-conquer method for on-device reduction (see section 4.4). Additionally, all the model parameters as well as fixed quantities such as gridsizes or convergence tolerances were placed in constant memory to facilitate *local* access to these values by each thread. No further optimizations were initially applied. The final code snippet shows some internals of the kernel code but omits device function. Internals of the device function are functionally identical to similar Fortran code and perform Gauss-Hermite integration of trilinearly interpolated value function.

Cuda kernel code

```
1  ...
2    __device__ inline double UpdateExpectedCTG(double u,double x,double mu,double Sigma,
3      double alpha, double gamma, double delta, double omega, double sigma_eps,
4      double xbar, double ubar, int NX,int Nmu, int NS, int NGH, double* XGrid,
5      double* muGrid, double* SigmaGrid, double* V, double* rno, double* wei);
6    __global__ void UpdateExpectedCTG_kernel(double* U,double* X,double* mu,
7      double* Sigma,double* V0,double* rno,double* wei,double* V1)
8      {
9
10     //Thread index
11     const int      tid = blockDim.x * blockIdx.x + threadIdx.x;
12     const int NUM_ITERATION= dc_NX*dc_Nmu*dc_NSigma;
13     int ix,jmu,kSigma;
14
15     //Total number of threads in execution grid
16     const int THREAD_N = blockDim.x * gridDim.x;
17
18     //ech thread works on as many points as needed to update the whole array
```

```
    for (int i=tid;i<NUM_ITERATION;i+=THREAD_N)
        {
            //update expected cost-to-go point-by-point
            ix=i/(dc_NSigma*dc_Nmu);
            jmu=(i-ix*dc_Nmu*dc_NSigma)/dc_NSigma;
            kSigma=i-ix*dc_Nmu*dc_NSigma-jmu*dc_NSigma;
            V1[i]=UpdateExpectedCTG(U[i],X[ix],mu[jmu],Sigma[kSigma],dc_alpha,
                dc_gamma,dc_delta,dc_omega,dc_sigmasq_epsilon,dc_xstar,dc_ustar,
                dc_NX,dc_Nmu,dc_NSigma,dc_NGH,X,mu,Sigma,V0,rno,wei);
        }

}
...
```

4.3. **Speed comparison.** Table 2 documents initial timing results for GPU implementations in single and double precision in comparison to single-threaded and multi-threaded CPU-based implementation. These are done for all three policies and across a range of gridsizes. As with CPU-based implementation, CUDA codes were timed repeatedly to reduce measurement noise.[14]

Several things are worth noticing in table 2. First is that across all cases, GPU wins over single CPU in 86% of cases, and over four cores in 94% of cases. GPU only loses for the smallest problem sizes, where multithreading, whether based on CPU or GPU, is actually detrimental to performance due to thread creation and destruction overheads. Second, the margin of victory is oftentimes quite substantial, in some cases exceeding a factor of 20. Third, single precision calculation on the GPU is generally faster than double precision, especially taking into consideration that dynamic programming algorithm takes up to 50% more iterations to converge in single precision, with exact ratio depending on gridsize and policy type. In contrast, the speed of single precision calculation on the CPU is only marginally different from double precision, after accounting for convergence effect. This is because GPUs have separate units dedicated to double and single precision floating point calculations but CPUs do not. Moreover, the generation of NVIDIA GPUs that we used has only 1/8 as many resources devoted to double precision as to single precision work. With this in mind, it is actually surprizing how little difference there is between double and single precision results on GPU. This is likely due to underutilization of floating point resources in either case.

To provide uniform basis for comparison we transformed timing results for the double precision case reported in table 2 into gridpoints per second speeds. The speeds are plotted in figure 1 against the overall size of the grid, in log space. For all three policy types, the performance of single-threaded code tends to fade with the problem size, most likely due to memory limitations. In contrast, the performance of two- and four-threaded versions of the two suboptimal policies initially improves with problem size as fixed costs of multiple threads are spread over longer runtime but starts to fall relatively early. More complex calculation per grid point for the optimal policy causes the downward trend in performance throughout the entire gridsize range. GPU performance, on the other hand, continues to improve until moderately large grid size but still drops for the very large grids.

Figure 2 distills performance numbers further by focusing on the GPU speedup ratio relative to the single CPU. It emphasizes two things – somewhat limited applicability of GPU speedups and substantial speed boost for moderately sized grids even for double precision.[15] Beyond the optimal problem size the gain in speed tapers off.[16]

[14]GPU runs have remarkably lower variability of measured runtime compared to CPU-based runs, possibly due to less intervention from the operating system.

[15]Similar but slightly slower results were reported in an earlier working paper version using CUDA 2.2, except for largest grids where computational acceleration faltered precipitously.

[16]Additionally, if GPU is used simultaneously to drive graphic display, GPU threads are currently limited to no more than 5 seconds of runtime each. Since lifetime of a thread in our implementation is one iteration of dynamic programming algorithm, and it takes 90-100 iterations to convergence, the total runtime is limited to about 500 seconds. To overcome this limitation one either has to use a dedicated graphics board or run the program without windowing system present.

TABLE 2. Runtime comparison of CPU and baseline GPU-based calculations.

Policy	**Gridsize**	**Grid Points**	**Single Precision**					**Double Precision**				
			CPU_1	CPU_4	GPU	$\frac{CPU_1}{GPU}$	$\frac{CPU_4}{GPU}$	CPU_1	CPU_4	GPU	$\frac{CPU_1}{GPU}$	$\frac{CPU_4}{GPU}$
Inert Uninformative Policy	8x8x8	512	0.114	0.723	0.231	0.49	3.13	0.096	0.27	0.216	0.46	1.25
	16x16x16	4,096	0.735	0.689	0.261	2.82	2.64	0.95	0.33	0.288	3.30	1.15
	32x32x32	32,768	7.039	2.669	1.027	6.85	2.60	8.42	2.71	1.070	7.87	2.53
	64x64x64	262,144	74.250	25.319	5.529	13.43	4.58	77.17	21.73	6.156	12.54	3.53
	128x128x128	2,097,152	748.119	223.696	42.282	17.69	5.29	798.45	211.18	49.630	16.09	4.26
	256x256x256	16,777,216	6,400.123	1,950.315	369.225	17.33	5.28	7,368.56	2,016.57	413.532	17.82	4.88
Cautionary Myopic Policy	8x8x8	512	0.09	0.521	0.211	0.43	2.47	0.094	0.22	0.270	0.35	0.81
	16x16x16	4,096	1.100	0.806	0.273	4.03	2.95	1.01	0.35	0.289	3.48	1.21
	32x32x32	32,768	11.397	4.056	1.150	9.91	3.53	9.72	3.01	1.588	6.13	1.90
	64x64x64	262,144	124.663	40.941	7.009	17.79	5.84	94.07	26.14	6.935	13.56	3.77
	128x128x128	2,097,152	1,218.964	383.088	59.939	21.50	6.39	962.46	270.37	64.693	14.88	4.18
	256x256x256	16,777,216	13,739.599	4,161.66	994.28	13.82	4.19	9,880.06	2,855.14	748.630	13.20	3.81
Optimal Policy	8x8x8	512	1.639	0.633	1.148	1.43	0.55	1.43	0.57	1.235	1.16	0.46
	16x16x16	4,096	16.105	5.287	1.528	10.54	3.46	13.22	4.79	3.493	3.78	1.37
	32x32x32	32,768	153.816	48.754	8.130	18.92	6.00	122.68	37.56	13.163	9.32	2.85
	64x64x64	262,144	1,413.794	422.316	64.576	21.89	6.54	1,085.47	344.43	84.756	12.81	4.06
	128x128x128	2,097,152	16,783.030	5,200.646	755.675	22.21	6.88	9,972.39	3,131.724	650.695	15.33	4.81
	256x256x256	16,777,216	202,209.075	61,810.338	16,054.990	12.59	3.85	98,253.32	29,972.30	6,966.786	14.10	4.30

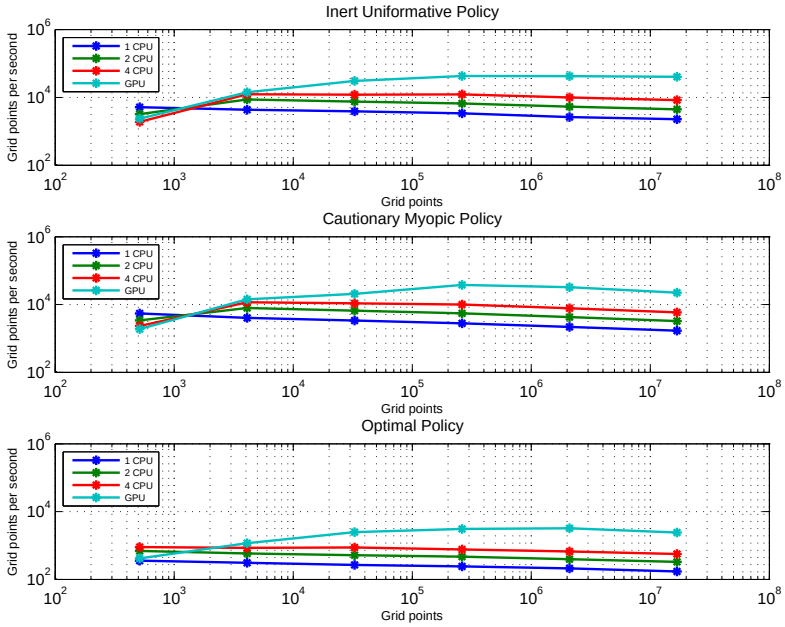

FIGURE 1. Speed comparison of CPU and GPU-based approaches for double precision calculations.

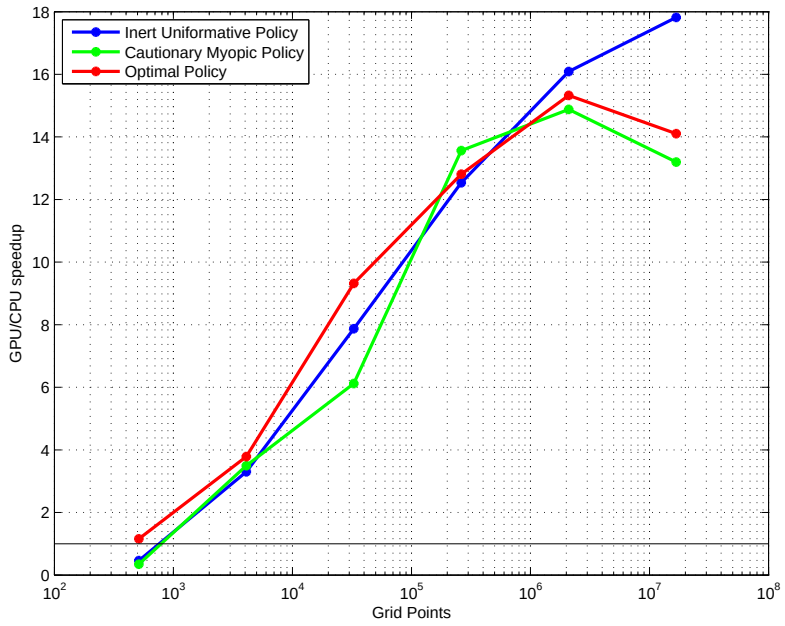

FIGURE 2. Ratio of GPU and single CPU speeds for double precision calculations.

4.4. **Code Tuning.** Substantial speed gains demonstrated above are a testament to the potential of throughput-oriented GPU architecture and resulted from straightforward code translation. Whether initial implementation leaves additional performance on the table is the focus of this section.

To identify potential performance bottlenecks of initial implementation we made use of CUDA profiler tool. Results for 64x64x64 problem configuration are reported in table 3.[17]

TABLE 3. CUDA Profiler report based on runs with 64x64x64 gridsize.

Policy	Computation	GPU Time	Average Occupancy	Divergent Branches	Registers per Thread
Inert Policy	GPU→CPU memory copy	0.148			
	Kernel Execution	5.400	18.8%	0.75%	71
	GPU→GPU memory copy	0.000			
	CPU→GPU memory copy	0.074			
Cautionary Myopic Policy	GPU→CPU memory copy	0.148			
	Kernel Execution	5.950	18.8%	5.21%	71
	GPU→GPU memory copy	0.000			
	CPU→GPU memory copy	0.074			
Optimal Policy	GPU→CPU memory copy	0.050			
	Kernel Execution	84.260	12.5%	3.83%	122
	GPU→GPU memory copy	0.007			
	CPU→GPU memory copy	0.001			

Ostensibly, execution time is dominated by kernel computations while explicit transfers between various memory spaces account for less than 3% of time. This is helpful since memory accesses are normally much slower than computations. Also favorable to performance is the low incidence of divergent branches. Divergent branches refer to situations when different threads within the same groups of threads called warp take different paths following a branch condition. Thread divergence leads to performance degradation.

On the other hand, average occupancy seems low. Average occupancy indicates that only 10-20% of threads (out of possible 30,720) can be launched simultaneously with enough per-thread resources. While occupancy does not necessarily equal performance, low occupancy kernels are not as good at hiding latency. NVIDIA recommends average latency of 25% or higher to hide latency completely (NVIDIA, 2009c). The limiting factor to occupancy is the number of registers, the fastest type of on-chip memory used to store intermediate calculations. More registers per thread will generally increase the performance of individual GPU threads. However, because thread registers are allocated from a global register pool, the more registers are allocated per thread will also reduce the maximum thread block size, thereby reducing the amount of thread parallelism (NVIDIA, 2009c). NVIDIA CUDA compiler provides a switch that allows control of the maximal number of registers per thread. If this

[17]Slight discrepancy in total times between tables 2 and 3 are due to small variation between individual runs and overheads of initialization and profiling counters.

option is not specified, no maximum is assumed. In addition, for a given register limit, occupancy may be tuned by manipulating execution configuration, i.e. the number of thread blocks and block size. For simplicity, we fixed execution configuration parameters at values suggested by CUDA occupancy calculator.

To assess the impact of specifying different maximum amount of registers per thread, we recompiled and rerun the code for the actively optimal control using register values from 20 to 128. 20 is the smallest amount that allows the program to run, and 128 is the maximum allowed by GT200 architecture. Results of that experiment, in terms of grid points per second, are shown in figure 3 for several problem sizes. The best results for each problem size are marked with solid circles and also collected in table 4.[18]

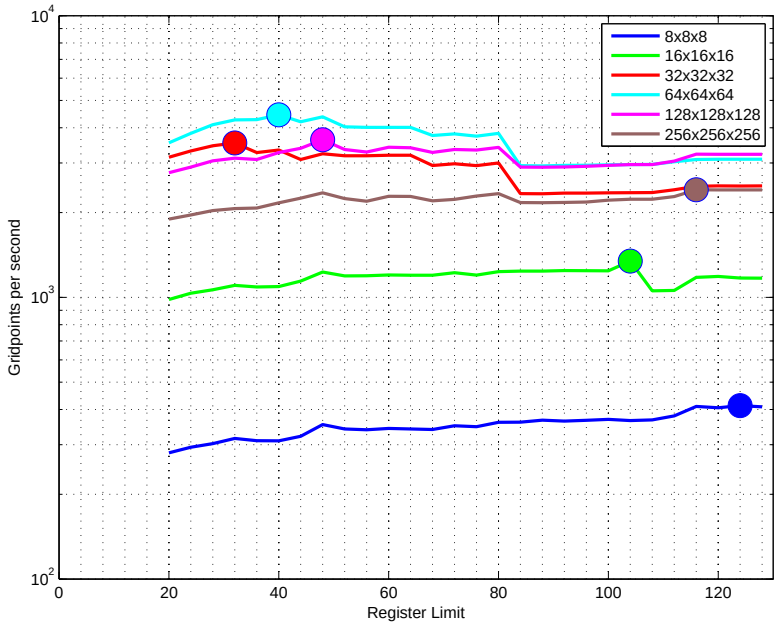

FIGURE 3. Register use and GPU speeds for active optimal policy calculation.

The good news in figure 3 and table 4 is ability to extract additional 40% of performance. The bad news is that tuning register use is not automatic but labor intensive. It also appears to be problem size specific. For smallest problems it is best to use almost all registers, while for medium-size problems it is enough to restrict register usage to 32 to 48 range. The largest problems again favor large number of registers. As scarcity of registers limits the amount of thread parallelism, this explains hump-shaped performance curve in figure 2. Since occupancy is in an inverse relationship with register use, the optimal occupancy exhibits inverse U shape with respect to the problem size. That occupancy is not equal performance is shown in figure 4. Occupancy declines throughout the entire range of register use limits, and yet the optimal register use is achieved in the interior. Still, the greatest performance boost is achieved for problem sizes with largest difference between baseline and optimized occupancy values.

Comparison of figure 3 and figure 4 suggests very strong correlation of performance and memory throughput reported by the CUDA profiler. It makes sense based on the following considerations.

Each thread in our implementation sequentially applies Bellman updates to one or more point on the state-space grid (more than one if total gridsize exceeds the number of available threads). For each update, the Gauss-Hermite quadrature requires access to as many off-the-grid points as there are quadrature knots (we use 32). Trilinear interpolation needs 8 on-the-grid points for each quadrature knot. Thus, updating a single grid point requires access to 256 values of the cost-to-go

[18]Similar results were obtained for the other two policies and thus not reported.

TABLE 4. Speedup due to register tuning for active optimal policy calculation.

Problem size	Default solution time	Optimal register restriction	Register-optimal solution time	Speedup factor	Optimal occupancy
8x8x8	1.253	124	1.241	0.95%	12.5%
16x16x16	3.493	92	3.291	6.14%	12.5%
32x32x32	13.163	32	9.273	41.95%	50%
64x64x64	84.756	40	58.961	43.75%	37.5%
128x128x128	650.695	48	578.393	12.50%	31.25%
256x256x256	6,966.786	120	6,966.982	0.00%	12.5%

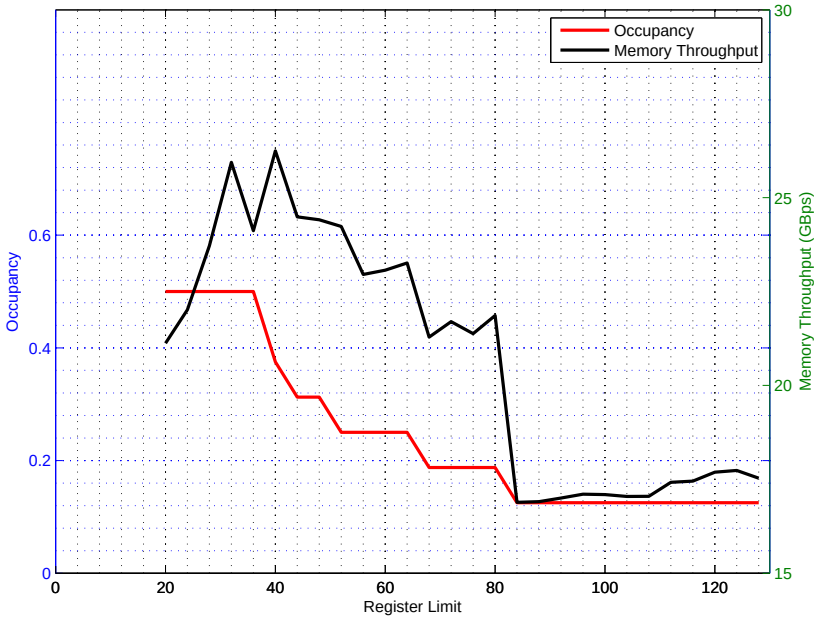

FIGURE 4. Occupancy and memory throughput against register use for active optimal policy calculation for 64x64x64 problem size.

function which is stored in global on-GPU memory.[19,20] For each such value, the kernel performs

[19]Technically, our algorithm features *gather* memory access type.

[20]Potentially, not all of these grid points are distinct, in which case there is loss of efficiency due to repeated yet unnecessary memory access. Adapting the code to track repetitions requires each thread to locally store potentially

four floating point operations resulting in *compute to global memory access ratio* (CGMA) of about 4. Low CGMA means that the memory access is indeed the limiting factor (Kirk and Hwu, 2010).

Since reported memory throughput falls short of the advertised theoretical maximum of 141.6 GB/sec, there are additional factors hindering performance. These factors have to do with *locality* of memory accesses. Indeed, the 256 values of the cost-to-go function needed to update given point could be widely spread out in the state space since they depend on the magnitude of control impulse under consideration. Therefore, Gauss-Hermite integration inside the value function update induces memory access pattern that is not random but somewhat irregular in that large and variable control impulse effectively spreads out integration nodes further apart so that access to more distant grid indices is required.[21] Since memory access is irregular, the popular strategy of storing subvolumes of the state space in constant memory will not work (Kirk and Hwu, 2010). Improving memory access requires more sophisticated approaches. One such approach is to do with textures.

Due to their graphics heritage, GPUs have dedicated hardware to process and store textures which can be thought of as a two-dimensional "skin" to be mapped onto a surface of a three-dimensional object. Formally, a texture can be any region in linear memory which is read-only and is cached, potentially exhibiting higher bandwidth if there is 3D locality in how it is accessed. Textures are read from kernels using device functions called *texture fetches*.[22] Due to special memory layout of texture fetches the codes with random or irregular memory access patterns typically benefit from binding multi-dimensional arrays to textures instead of storing them in global on-device memory. Thus, texture fetches can provide higher memory bandwidth to computational kernels.[23] With this in mind, we bound our policy and value arrays to one-dimensional textures and used texture fetches inside the CUDA kernel. Results are shown in figure 5 for the baseline case with unlimited per-thread registers in the left panel and for the register-tuned case in the right panel.

The inert uninformative policy features regular access pattern to one-dimensional arrays storing value and policy functions. Thus, it doesn't benefit much from texture memory access. Under the cautionary myopic policy, using textures can be highly beneficial, especially for larger problem sizes. Calculations for optimal policy are also consistently accelerated by texture fetches for larger problems, but for smaller problems it could lead to a performance loss.

An important performance recommendation according to NVIDIA (2009c) is to minimize data transfers between the host and the device because those transfers have much lower bandwidth than internal device data transfers. In our initial implementation, convergence check was done on the CPU which necessitates transfer of data out of GPU device. Convergence check amounts to calculating the maximum difference between two arrays, and is thus a reduction (Kirk and Hwu, 2010). Reduction step could be performed on GPU. We extended our computational kernel to accommodate on-device partial reduction in which each thread block computes the maximal absolute difference for array indices assigned to that thread block. This is done by traversing a comparison tree in a way that aligns memory accesses by each thread, unrolls loops and uses shared memory. Partial reduction code was derived from Harris (2005). It has high-performance on large arrays but is rather complex and repetitive, especially for the optimal policy calculations which are hampered by CUDA's restriction on function pointers that prevent us from putting optimization code into a separate function. Once the first thread in a thread block finds the max norm distance between successive value function approximations for the slice assigned to that threadblock, the second round of reduction commences where a small subset of threads is used to compute the largest of threadblock maxima. This step

large number of values thus increasing register pressure and complicating code flow. For these two reasons we decided against access tracking.

[21] Do-nothing policy is an exception since it leaves beliefs and integration nodes unchanged.

[22] Texture fetches are not available in CUDA Fortran.

[23] In addition, texture fetches are further capable of hardware-accelerated fast low-precision trilinear interpolation capabilities with automatic handling of boundary cases. This could be very useful since value and policy function are approximated by trilinear interpolation in between grid points. The problem, as of current implementation of CUDA and GPU hardware is lack of support for double precision texture fetches. Accessing double precision memory can be overcome by using union of double and integer, but double precision interpolation remains impossible. Also, it's been observed that effectiveness of textures is highly dependent on CUDA hardware and driver, with some "magic" access patterns much faster than others Barros (2009); Barros, Babich, Brower, Clark, and Rebbi (2008).

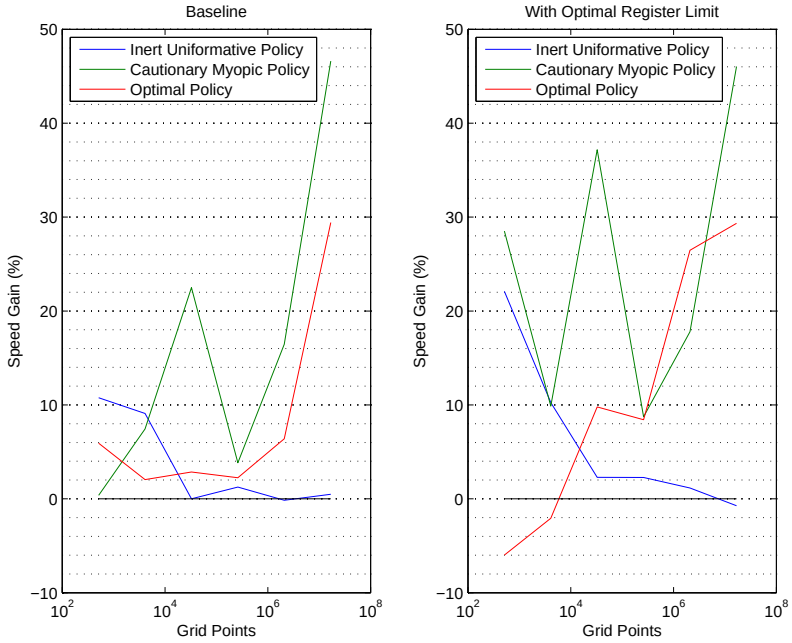

FIGURE 5. Speed gains due to accessing memory via textures.

is short since there could be no more than 512 threadblocks. We have not taken up suggestion by Boyer, Tarjan, Acton, and Skadron (2009) to reorder reduction and computation steps to eliminate wasteful synchronization barrier between first and second rounds.

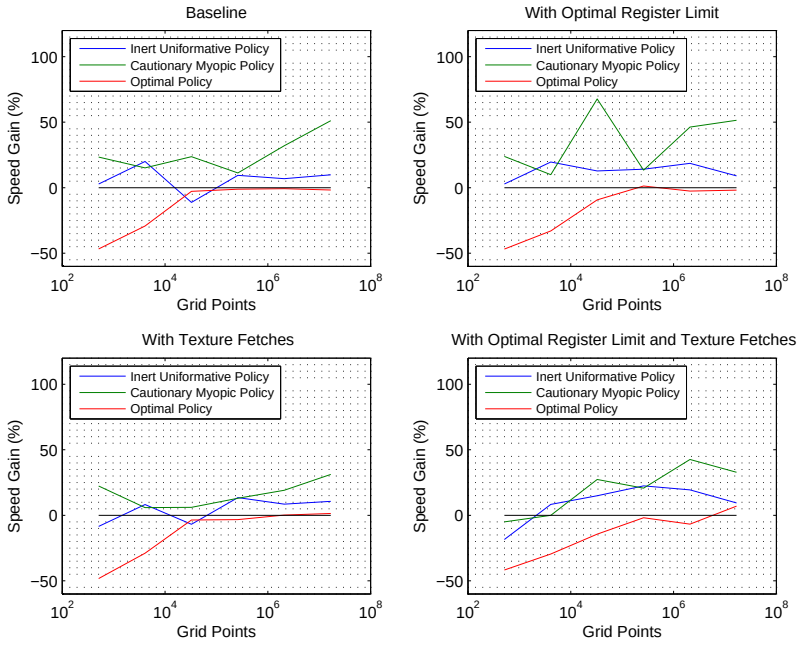

FIGURE 6. Speed gains due to on-device partial reduction.

Figure 6 summarizes performance impact due to on-device reduction in combination with other methods. On-device reduction is not always beneficial, especially for small problem sizes as well

as for more complex algorithms. For example, optimal policy computation is accelerated only in combination with other techniques and for the largest problem sizes, since it is largely compute-bound, not memory-bound. On the other hand, computing the value of the cautionary myopic policy profits from on-device reduction practically across the board. In combination with register tuning, it can gain impressive 68%. The speed gain tends to diminish when on-device reducton is in addition to texture fetches since both techniques aim to alleviate the same memory bottleneck.

CUDA offers several additional techniques and tools that may help performance in some applications. These include page-locked memory (*pinned memory*) mapped for access by GPU for higher transfer speeds and eligibility for asynchronous memory transfers as well as *atomic functions* that are guaranteed to be performed without interference from other threads. We investigated these techniques and found neither speed advantage nor disadvantage for our code.

Table 5 assembles our best speedups relative to a single CPU for double precision code and how to achieve them. It shows that we are able to extract up to 26 times more speed from a graphics device than from a single thread running on high end CPU. It also highlights that achieving maximal perfomance is not straightforward but comes from a search over multiple performance dimensions. It is doubtful that researchers will have enough time on their hands to do much beyond simple tune-ups.

For the smallest problem sizes GPU computing is not worth the effort and can be even counter-productive.

5. Concluding Remarks

This paper offers qualified endorsement of data parallel massively multi-threaded computation using CUDA for computational economics. With relatively little effort, we were able to boost performance by a factor of about 15 relative to optimized single-thread CPU implementation. Nevertheless, attaining full potential of graphics hardware proved more involved. The nuisance is that for parallel programming on the current generation of graphics chips one needs to know certain details of how the underlying architecture actually works. Notably, one has to heed the management of thread hierarchy by balancing size and number of blocks, by hiding memory latency through overlapping inter-GPU communication with computation, by explicitly optimizing memory access patterns across different memory types and different memory locations. Appropriate combinations of performance tuning techniques, which essentially trade one resource constraint for another, can make a huge difference in the final performance. Plausible strategies to optimize around various performance gating factors embrace ideas of maximizing independent parallelism, workload partitioning and tiling; loop unrolling; using shared, constant and texture memories; data prefetch; asynchronous data transfers and optimizing thread granularity (see NVIDIA (2009c)). Since in different applications different resource limits may be binding, performance tuning is prone to dull guess work.[24] As a corollary, the dearth of high-level auto-tuning tools frustrates development effort, engenders coding errors and inhibits scalability across future generations of hardware.

GPU-based numerical libraries are in their infancy as well. While basic dense linear algebra, some signal processing (e.g., FFT) and elementary random number generation are covered (NVIDIA, 2009a; Tomov, Dongarra, Volkov, and Demmel, 2009; NVIDIA, 2009b; Howes and Thomas, 2007), important areas such as multivariate optimization and quadrature are not. Integration with different programming languages such as `Fortran` or `Java` or with computing environments such as `Matlab` or `Mathematica` is also immature. These areas need to improve before GPU computation can become truly popular.

The tedium of debugging parallel code detracts from GPU programming even more. Parallel code begets new breed of errors including resource contention, race conditions and deadlocks. Since even very unlikely event will likely occur given sufficiently many trials, the severity of these errors rapidly escalates with the number of parallel threads. For example, triggering a subtle synchronization bug

[24] Kirk and Hwu (2010) report in one their case studies that exhaustive search over various tuning parameters and code organizations resulted in 20% larger performancce increase than was available by exploiting heuristic considerations.

TABLE 5. Maximal speedups.

Policy	Gridsize	Maximal Speedup	Register Limit	Textures	On-device Reduction
Inert Uninformative Policy	8x8x8	0.58	116	yes	yes
	16x16x16	3.96	unlimited	no	yes
	32x32x32	9.91	40	yes	yes
	64x64x64	16.74	40	yes	yes
	128x128x128	21.34	36	yes	yes
	256x256x256	22.78	32	yes	yes
Cautionary Myopic Policy	8x8x8	0.45	120	yes	no
	16x16x16	4.02	84	yes	no
	32x32x32	10.72	40	yes	yes
	64x64x64	18.58	40	yes	yes
	128x128x128	25.19	36	yes	yes
	256x256x256	26.25	72	yes	yes
Optimal Policy	8x8x8	1.21	unlimited	yes	no
	16x16x16	4.02	92	no	no
	32x32x32	14.52	32	yes	no
	64x64x64	19.96	40	yes	no
	128x128x128	21.80	48	yes	no
	256x256x256	19.54	48	yes	yes

may be extremely unlikely with small number of concurrent threads, that same bug will have a much higher probability of manifestation in a parallel program with tens of thousands of threads. Even so, uncovering and solving a bug in a parallel program will not necessarily get easier with higher likelihood of bug's appearance. This is because effective reasoning about parallel workflow is difficult even for seasoned programmers. While a computational economist who is either adventurous or desperate for performance will have to balance all these issues against very substantial potential gains, it does appear that the effort may be worth it.[25]

[25]One should also be aware that claimed speed gains of over 100x are unrealistic unless CPU implementation is performance handicapped (Lee, Kim, Chhugani, Deisher, Kim, Nguyen, Satish, Smelyanskiy, Chennupaty, Hammarlund, Singhal, and Dubey, 2010). For a fair comparison, both CPU and GPU implementations need to be fine-tuned, a potentially difficult task in both cases.

Development of parallel computing on GPUs is bound to have an impact on high performance computing industry as graphics cards are already inexpensive and ubiquitous. Indeed, with peak performance of nearly one teraflop, the compute power of today's graphics processors dwarfs that of the commodity CPU while costing only a few hundred dollars. If nothing else, emergence of GPU as a viable competitor to traditional CPU for high performance computing will elicit a response by major CPU manufacturers, a duopoly of Intel and AMD. Even though CPU development may appear positively glacial compared to quantum performance leaps in GPUs, nothing prevents CPUs from adopting ideas of data parallel processing. CPUs are growing to have more cores, capable of more threads, add special units for streaming and vector computations. Already, many-core CPU and GPU hybrids are in development.[26]

GPU architecture and capabilities are still undergoing rapid development. For instance, the next generation of NVIDIA GPUs and supporting software (NVIDIA Parallel Nsight), released in early 2010, mitigates the performance disadvantage of double precision floating point arithmetics, enables larger 64-bit memory address space which is unified across host CPU and GPU devices, allows concurrent execution of heterogeneous kernels, lifts a number of C/C++ language restrictions and provides better tools for debugging and performance analysis (Kirk and Hwu, 2010). In a way, GPUs continue to become more general purpose, while CPUs acquire special purpose GPU-like components.

The relative balance between using moderate number of general purpose CPU-like components and large number of special purpose GPU-like components in future designs is not clear, and will likely to vary over time and among different device makers. Either way, there seems to be no doubt that future generations of computer systems, ranging from handheld devices to supercomputers, will consist of a composition of heterogeneous components. Furthermore, it doesn't matter whether GPUs will merge into CPUs or vice versa. What does matter is how to harness the raw power of heterogenous task-parallel and data-parallel devices for general purpose computations. Development of programming tools is essential here and the effort is already well under way.[27] Thus, our somewhat bleak assessment of writing fast parallel code as a black magic, mastered only by those stubborn enough to persevere through a long journey of frustration, may yet to be upgraded.

The future of many computations belongs to parallel algorithms. Today's era of traditional von Neumann sequential programming model (Backus, 1977) with its escalating disparity between the high rate at which CPU can work and limited data throughput between it and memory is nearly over. Modern processors are becoming wider but not faster. The change in the computational landscape presents both challenges and opportunities for the economists and financial engineers. In this regard, our paper is but a modest attempt to pick up low hanging fruit. Further case studies to cultivate intuition about the types of algorithms that can result in high perfomance execution on massively parallel co-processor devices as well as economic applications that can benefit from that performance are needed.

[26] For example, Intel plans to enter GPGPU and high performance computing markets with its own many-core device, called Intel Many Integrated Core (MIC) architecture and is said to build on an earlier Larrabee design (Seiler, Carmean, Sprangle, Forsyth, Abrash, Dubey, Junkins, Lake, Sugerman, Cavin, Espasa, Grochowski, Juan, and Hanrahan, 2008). This new architecture is promised to resemble the well known programming model for x86 multi-core architectures, so that many C/C++ applications can be simply recompiled for MIC and execute correctly with no code modification. This application portability, together with ability to hide low-level details from programmer, could produce large performance gains without extreme algorithmic effort. Whether this promise is realized remains to be seen.

[27] Among other things, compiler vendors are working on inclusion of GPU accelerator technology with their Fortan and C compilers. For example, using provisional support by PGI Group (The Portland Group, 2008), programmers will be able to accelerate Linux applications by adding OpenMP-like compiler directives that instruct the compiler to analyze the whole program structure and data, split portions of the applications between CPU and GPU as specified by user directives, and define and generate an optimized mapping of loops to automatically use the parallel cores, hardware threading capabilities and SIMD vector capabilities of modern GPUs.

References

Abdelkhalek, A., A. Bilas, and A. Michaelides (2001): "Parallelization, optimization and performance analysis of portfolio choice models," in *Proceedings of the 2001 International Conference on Parallel Processing (ICPP01)*.

Backus, J. (1977): "Can Programming be Liberated from the von Neumann Style?," *Communications of the ACM*, 21(8), 613–641.

Bargen, B., and P. Donnelly (1998): *Inside DirectX*, Microsoft Programming Series. Microsoft Press, Redmond, Washington.

Barros, K. (2009): "CUDA Tricks and Computational Physics," Guest lecture, Course 6.963, Massachusetts Institute of Technology.

Barros, K., R. Babich, R. Brower, M. A. Clark, and C. Rebbi (2008): "Blasting through lattice calculations using CUDA," Discussion paper, The XXVI International Symposium on Lattice Field Theory, Williamsburg, Virginia, USA.

Beck, G. W., and V. Wieland (2002): "Learning and control in a changing economic environment," *Journal of Economic Dynamics and Control*, 26(9-10), 1359–1377.

Bennemann, C., M. W. Beinker, D. Eggloff, and M. Guckler (2008): "Teraflops for Games and Derivatives Pricing," *Wilmott Magazine*, pp. 50–54.

Bertsekas, D. P. (2001): *Dynamic Programming and Optimal Control*, vol. 2. Athena Scientific, Nashua, NH, 2 edn.

——— (2005): *Dynamic Programming and Optimal Control*, vol. 1. Athena Scientific, Nashua, NH, 3 edn.

Boyd, C., and M. Schmit (2009): "Direct3D 11 Compute Shader," WinHEC 2008 presentation.

Boyer, M., D. Tarjan, S. T. Acton, and K. Skadron (2009): "Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors," in *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy.

Brent, R. P. (1973): *Algorithms for Minimization without Derivatives*. Prentice-Hall, Englewood Cliffs.

Brezzia, M., and T. L. Lai (2002): "Optimal learning and experimentation in bandit problems," *Journal of Economic Dynamics and Control*, 27(1), 87–108.

Buck, I. (2005): "Stream computing on graphics hardware," Ph.D. Dissertation, Stanford University, Stanford, CA, USA.

Chandra, R., R. Menon, L. Dagum, and D. Kohr (2000): *Parallel Programming in OpenMP*. Morgan Kaufmann.

Chapman, B., G. Jost, and R. van der Paas (2007): *Using OpenMP: Portable Shared Memory Parallel Programming*, Scientific and Engineering Computation Series. MIT Press, Cambridge, MA.

Chong, Y. Y., and D. F. Hendry (1986): "Econometric Evaluation of Linear Macro-economic Models," *The Review of Economic Studies*, 53(4), 671–690.

Coleman, W. J. (1992): "Solving Nonlinear Dynamic Models on Parallel Computers," Discussion Paper 66, Institute for Empirical Macroeconomics, Federal Reserve Bank of Minneapolis.

Conn, A. R., N. I. M. Gould, and P. L. Toint (1997): "A Globally Convergent Augmented Lagrangian Barrier Algorithm for Optimization with General Inequality Constraints and Simple Bounds," *Mathematics of Computation*, 66(217), 261–288.

Creel, M. (2005): "User-Friendly Parallel Computations with Econometric Examples," *Computational Economics*, 26(2), 107–128.

Creel, M., and W. L. Goffe (2008): "Multi-core CPUs, Clusters, and Grid Computing: A Tutorial," *Computational Economics*, 32(4), 353–382.

Doornik, J. A., D. F. Hendry, and N. Shephard (2002): "Computationally-intensive econometrics using a distributed matrix-programming language," *Philosophical Transactions of the Royal Society of London, Series A*, 360, 1245–1266.

Doornik, J. A., N. Shephard, and D. F. Hendry (2006): "Parallel Computation in Econometrics: A Simplified Approach," in *Handbook of Parallel Computing and Statistics*, pp. 449–476.

Chapman & Hall/CRC.

Easley, D., and N. M. Kiefer (1988): "Controlling a Stochastic Process with Unknown Parameters," *Econometrica*, 56(5), 1045–1064.

Ferrall, C. (2003): "Solving Finite Mixture Models in Parallel," Computational Economics 0303003, EconWPA.

Ghuloum, A., E. Sprangle, J. Fang, G. Wu, and X. Zhou (2007): "Ct: A Flexible Parallel Programming Model for Tera-scale Architectures," White paper, Intel Corporation.

Goffe, W. L., G. D. Ferrier, and J. Rogers (1994): "Global optimization of statistical functions with simulated annealing," *Journal of Econometrics*, 60(1-2), 65–99.

Goldberg, D. E. (1989): *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA.

Gropp, W., E. Lusk, A. Skjellum, and R. Thakur (1999): *Using MPI: Portable Parallel programming with Message Passing Interface*, Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 2 edn.

Harris, M. (2005): "Mapping Computational Concepts to GPUs," in *GPU Gems*, ed. by M. Parr, vol. 2, chap. 31, pp. 493–500. Addison-Wesley.

Horst, R., and P. M. Paradalos (eds.) (1994): *Handbook of Global Optimization*, vol. 1 of *Nonconvex Optimization and Its Applications*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Howes, L., and D. Thomas (2007): "Efficient Random Number Generation and Application Using CUDA," in *GPU Gems 3*, ed. by H. Nguyen, chap. 37, pp. 805–830. Addison-Wesley Professional.

Judge, G. G., T.-C. Lee, and R. C. Hill (1988): *Introduction to the Theory and Practice of Econometrics*. Wiley, New York, 2 edn.

Kendrick, D. A. (1978): "Non-convexities from probing an adaptive control problem," *Journal of Economic Letters*, 1(4), 347–351.

Kessenich, J., D. Baldwin, and R. Rost (2006): *The OpenGL Shading Language*. Khronos Group.

Khronos OpenCL Working Group (2009): *The OpenCL Specification, Ver. 1, Rev. 48*. Khronos Group.

Kirk, D. B., and W. W. Hwu (2010): *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan-Kaufmann.

Kirkpatrick, S., C. D. Gelatt Jr., and M. P. Vecchi (1983): "Optimization by Simulated Annealing," *Science*, 220(4598), 671–680.

Kola, K., A. Chhabra, R. K. Thulasiram, and P. Thulasiraman (2006): "A Software Architecture Framework for On-line Option Pricing," in *Proceedings of the 4th International Symposium on Parallel and Distributed Processing and Applications (ISPA-06)*, vol. 4330 of *Lecture Notes in Computer Science*, pp. 747–759, Sorrento, Italy. Springer-Verlag.

Lagarias, J. C., J. A. Reeds, M. H. Wright, and P. E. Wright (1998): "Convergence Properties of the Nelder-Meade Simplex Method in Low Dimensions," *SIAM Journal of Optimization*, 9(1), 112–147.

Lee, V. W., C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey (2010): "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," *SIGARCH Comput. Archit. News*, 38(3), 451–460.

Lewis, R. M., and V. Torczon (2002): "A Globally Convergent Augmented Lagrangian Pattern Search Algorithm for Optimization with General Constraints and Simple Bounds," *SIAM Journal on Optimization*, 12(4), 1075–1089.

Lindoff, B., and J. Holst (1997): "Suboptimal Dual Control of Stochastic Systems with Time-Varying Parameters," mimeo, Department of Mathematical Statistics, Lund Institute of Technology.

Morozov, S. (2008): "Learning and Active Control of Stationary Autoregression with Unknown Slope and Persistence," Working paper, Stanford University.

——— (2009a): "Bayesian Active Learning and Control with Uncertain Two-Period Impulse Response," Working paper, Stanford University.

——— (2009b): "Limits of Passive Learning in the Bayesian Active Learning Control of Drifting Coefficient Regression," Working paper, Stanford University.

Nagurney, A., T. Takayama, and D. Zhang (1995): "Massively parallel computation of spatial price equilibrium problems as dynamical systems," *Journal of Economic Dynamics and Control*, 19(1-2), 3–37.

Nagurney, A., and D. Zhang (1998): "A massively parallel implementation of discrete-time algorithm for the computation of dynamic elastic demand and traffic problems modeled as projected dynamical systems," *Journal of Economic Dynamics and Control*, 22(8-9), 1467–1485.

Nelder, J. A., and R. Mead (1965): "A simplex method for function minimization," *Computer Journal*, 7, 308–313.

NVIDIA (2008): *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide* NVIDIA Corporation, Santa Clara, CA, 2 edn.

NVIDIA (2009a): *CUBLAS Library.* NVIDIA Corporation, Santa Clara, CA, 2.3 edn.

——— (2009b): *CUFFT Library.* NVIDIA Corporation, Santa Clara, CA, 2.3 edn.

——— (2009c): *NVIDIA CUDA C Programming Best Practices Guide.* NVIDIA Corporation, Santa Clara, CA 95050, CUDA Toolkit 2.3 edn.

Paradalos, P. M., and H. E. Romeijn (eds.) (2002): *Handbook of Global Optimization*, vol. 2 of *Nonconvex Optimization and Its Applications*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Pflug, G. C., and A. Swietanowski (2000): "Selected parallel optimization methods for financial management under uncertainty," *Parallel Computing*, 26(1), 3–25.

Porteus, E. L., and J. Totten (1978): "Accelerated Computation of the Expected Discounted Returns in a Markov Chain," *Operations Research*, 26(2), 350–358.

Prescott, E. C. (1972): "The Multi-Period Control Problem Under Uncertainty," *Econometrica*, 40(6), 1043–1058.

Rahman, M. R., R. K. Thulasiram, and P. Thulasiraman (2002): "Forecasting Stock Prices using Neural Networks on a Beowulf Cluster," in *Proceedings of the Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, ed. by S. Akl, and T.Gonzalez, pp. 470–475, Cambridge, MA USA. IASTED, MIT Press.

Sanders, J., and E. Kandrot (2010): *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, Upper Saddle River, NJ.

Seiler, L., D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan (2008): "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics*, 27(3), 15 pages.

Sims, C. A., D. F. Waggoner, and T. Zha (2008): "Methods for inference in large-scale multiple equation Markov-switching models," *Journal of Econometrics*, 142(2), 255–274.

Spendley, W., G. R. Hext, and F. R. Himsworth (1962): "Sequential application of simplex designs in optimization and evolutionary design," *Technometrics*, 4, 441–461.

Svensson, L. E. O. (1997): "Optimal Inflation Targets, 'Conservative' Central banks, and Linear Inflation Contracts," *American Economic Review*, 87(1), 96–114.

Swann, C. A. (2002): "Maximum Likelihood Estimation Using Parallel Computing: An Introduction to MPI," *Computational Economics*, 19(2), 145–178.

The Portland Group (2008): *PGI Fortran & C Accelerator Compilers and Programming Model Technology Preview.* The Portland Group, Version 0.7.

Tomov, S., J. Dongarra, V. Volkov, and J. Demmel (2009): *MAGMA Library.* University of Tennessee, Knoxville, and University of California, Berkeley.

Wieland, V. (2000): "Learning by Doing and the Value of Optimal Experimentation," *Journal of Economic Dynamics and Control*, 24(4), 501–535.

Zabinsky, Z. B. (2005): *Stochastic Adaptive Search for Global Optimization*. Springer.

Zenios, S. A. (1999): "High-performance computing in finance: The last 10 years and the next," *Parallel Computing*, 25(13-14), 2149–2175.

E-mail address: Sergei.Morozov@morganstanley.com,mathur.sudhanshu@gmail.com