Massively Parallel Computation Using Graphics Processors with Application to Optimal Experimentation in Dynamic Control

Sudhanshu Mathur and Sergei Morozov

Morgan Stanley, New York

Computing in Economics and Finance, 2009

Talk outline

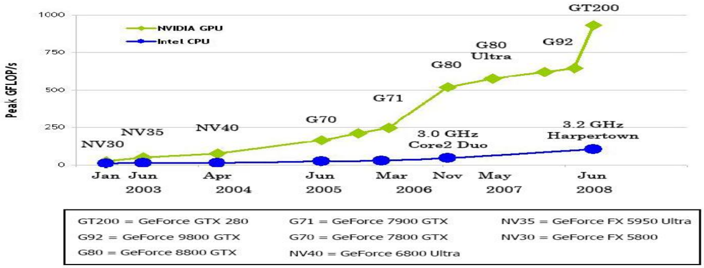
From gaming to high performance scientific computing

- Emergence of GPU as supercomputers on plug-in boards
- Coding for NVIDIA GPUs
- Case study: Dynamic programming solution of learning and active experimentation problem
- Future trends

What is GPU?

- dedicated graphics rendering device
- attached to plug-in board (video card) or directly to motherboard
- does calculations related to 3D computer graphics
- 3D computer graphics is based on matrix and vector operations

GPUs are fast and getting faster



NVidia GTX280:

- has more transistors than people in China (1.4 Bn)
- can process 30,720 threads simultaneously
- 933 Gflops in single precision
- has memory bandwidth of 141.7 GB/sec

Why are GPUs so fast?

Commoditization of parallel computing

- GPUs specialize in data-parallel computation
- More transistors devoted to data processing instead of data caching and flow control
- Commodity industry & economies of scale
    - gaming and entertainment
    - over 100 million units shipped since mid-2007
    - over 10 exaflops of sustained aggregate hardware performance
- Competitive industry fuels innovation
    - triumvirate ATI, NVIDIA and Intel

Pros and Cons

- Advantages
  - Speed
  - Programmability
  - Low cost
  - Massively parallel programming is inevitable anyway
  - Abstracted a layer above "metal"
- Difficulties
  - Programming model
    - unusual
    - tightly constrained (e.g. explicit memory management)
    - hard to debug (race conditions, locks) and validate (floating point arithmetic is not associative)
  - Rapidly evolving feature set
  - Double precision is significantly slower and only available since mid-2008
  - Limited on-board memory
  - Proprietory/secret underlying architecture

Evolution of GPU programming model

- Purely for graphics: OpenGL, DirectX
- GPGPU - stream processing
- Nvidia's CUDA interface library
- OpenCL - similar to CUDA but just for Nvidia

Non-gaming applications

- Numerics
  - random number generation
  - linear algebra
  - fast Fourier transform
- Physics
  - computational fluid dynamics
  - multi-body astrophysics
  - general relativistic evolution
  - weather forecasting
- Computer science
  - Computer vision, pattern and speech recognition
  - Cryptography
  - Electronic design automation
- Life sciences
  - protein folding
  - biomedical image analysis
  - artificial neural circuit simulations
  - DNA sequencing
- Finance
  - option pricing
  - risk analysis and algorithmic trading

General purpose GPU Programming

Data parallel programming vs Task parallel programming

- Task parallel: threads have their own goal and task
- Data parallel: same block of code is run over multiple data points

NVIDIA CUDA

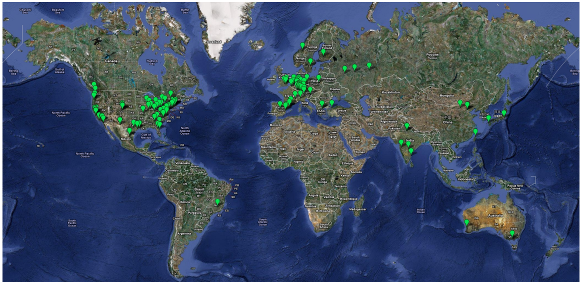
Compute Unified Driver Architecture

- Allows heterogeneous computation mixing code for CPU and GPU
- Based on shared memory model without explicit thread management
  - like OpenMP
- Consists of
  - runtime and function libraries
  - C/C++ development kit
  - extensions to C programming language
  - hardware abstraction mechanism
- CUDA-capable devices
  - GeForce 8 and newer
  - Tesla

Learning NVIDIA CUDA

Compute Unified Driver Architecture

- CUDA is taught in universities around the world



Computational economics case study

Dynamic programming solution of learning and active experimentation problem

- Active learning problems require brute-force
- Dynamic programming is economics workhorse

Imperfect Information Control Problem

- Objective

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

- subject to

$$\text{observed state: } x_t = \alpha + \beta u_t + \gamma x_{t-1} + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}\left(0, \sigma_\epsilon^2\right).$$

- Unknown β are characterized by prior and posterior beliefs
  - Prior

$$p(\beta) = \mathcal{N}(\mu_0, \Sigma_0)$$

  - Posterior

$$p(\beta|\mathcal{F}_t) = p(\beta|\{x_j, u_j\}_{j=1}^{t}) = \mathcal{N}(\mu_t, \Sigma_t)$$

- Known: state evolution parameters $\alpha, \gamma \in (-1, 1)$, σ_ϵ^2 and preference parameters $\delta \in (0, 1)$, $\omega > 0$, $\bar{x}$, $\bar{u}$.
- Stylized representation of problems under parameter/model uncertainty
  - monetary/fiscal stabilization; exchange rate targeting; pricing of government debt; trade policy
  - monopolistic pricing with unknown demand
  - natural resource extraction ...

Complete State Space Description

- Bayesian learning dynamics over location and scale:

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

- Information state as of end-of-date t: $\mathcal{I}_t = (\mu_{t+1}, \Sigma_{t+1})'$
- Information is endogenous state
- Extended state $\mathcal{S} = \mathcal{X} \times \mathcal{I} \subseteq \mathbb{R}^3$
- Bayesian updating and observed state evolution form nonlinear mapping on extended state $B(\cdot, x_{t-1}, u_t) : \mathcal{S} \to \mathcal{S}$

Dynamic Programming Formulation

- Stationary Bellman Equation for continuation value (cost-to-go)

$$
\begin{aligned}
V(S_t) = \min_{\{u_{t+1}\}} \Big\{ & L(S_t, u_{t+1}) \\
& + \delta \int V\left(B(S_t, \alpha + \beta u_{t+1} + \gamma x_t + \epsilon_{t+1}, u_{t+1})\right) p(\beta | S_t) q(\epsilon_{t+1}) d\beta d\epsilon_{t+1} \Big\} \\
& =: T[V](S_t),
\end{aligned}
$$

  where $L(S_t, u_t)$ is expected one-period loss and $T[V]$ is Bellman functional operator
- Exploration (learning) vs Exploitation (stabilization)
- T is a contraction mapping, value function iterations converge (Kiefer-Nyarko, 1989)
- Solution is two *functions*: optimal policy rule $u^* : \mathcal{S} \to \mathbb{R}$ and corresponding cost-to-go function $V : \mathcal{S} \to \mathbb{R}^+$

Why Is This Class of Problems Difficult?

- Bayes law is nonlinear
- State dimension increases rapidly with number of unknowns ("this method founders on the reef of dimensionality", R. Bellman 1956)
- Cost-to-go function need not be convex
- Policy function may have discontinuities
- Optimal cost-to-go function may have kinks
- Methods that rely on smoothness may fail in some parts of state: projection, perturbation, non-adaptive Smolyak sparse grids
- Unbounded state space

Simple Policy Alternatives

- Do-nothing policy: $u \equiv 0$
  - a.k.a *inert uniformative policy*
- Cautionary myopic policy
  - optimizes one-period-ahead expected loss
- Alternatives are useful for
  - benchmarking value of experimentation
  - test numerical codes for correctness without focus on policy optimality

Inert Uninformative Policy

Do-nothing policy

- Leaves posterior beliefs unchanged
- Cost-to-go function satisfies functional recursion

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

- Has closed-form cost-to-go function, quadratic in x
- Functional recursion can be used for approximate CPU- and GPU-based computation, while closed form gives correctness check[1]
- Yields analytic bounds on optimal policy

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

[1] Beware of edge effects!

Cautionary Myopic Policy

Optimize one-period-ahead expected loss

- $u_{t+1}^{MYOP}(x_t, \mu_{t+1}, \Sigma_{t+1}) = -\frac{\mu_{t+1}\gamma}{\Sigma_{t+1}+\mu_{t+1}^2+\omega} x_t + \frac{\mu_{t+1}(\bar{x}-\alpha)+\omega\bar{u}}{\Sigma_{t+1}+\mu_{t+1}^2+\omega}$
- u_{t+1}^{MYOP} is at the mid-point of analytic bounds on optimal policy
- Actual cost-to-go under cautionary myopic policy satisfies

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

- μ', Σ' evolve nonlinearly via Bayes rule
- There is no closed form solution for $V^{MYOP}(x,\mu,\Sigma)$
- Functional recursion can be iterated to convergence

CPU-based Computation

Sophisticated brute-forcing

- Initial state box chosen by suboptimal policy simulations
- State box expanded until reflective boundary effects are small
- Multilinear interpolation on non-uniform product grid (discontinuities are near $x_t = \bar{x}$ and $\mu = 0$)
- Parallel synchronous Gauss-Jacobi policy iteration for cost-to-go function approximation for do-nothing and cautionary myopic policies until 1e-6 relative error
- Parallel synchronous Gauss-Jacobi value iteration for cost-to-go function approximation for optimal policy until 1e-4 relative error
- Safeguarded univariate optimization with multiple starting points
- Fortran 90 with OpenMP on overclocked shared memory quadcore Core i7 with 8 GB RAM
- 40 min to 60 hrs depending on number of CPUs and policy for large grid with $\delta = \gamma = 0.9$, $\sigma_\epsilon^2 = \omega = x^* = 1.0$, $\alpha = u^* = 0$

Scaling of CPU-based Computation

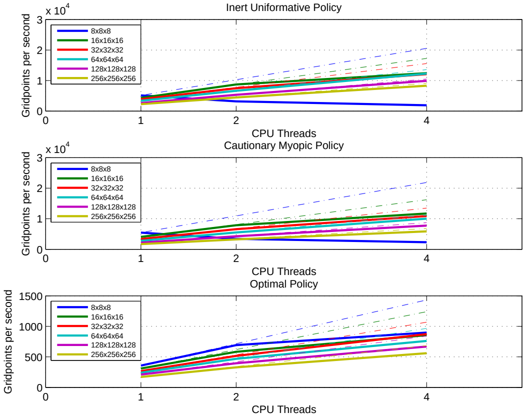
The problem is sufficiently parallel

- Scales almost linearly with number of processors
- Multiprocessing can be counterproductive at small problem sizes
- Memory can become a limiting factor at large problem sizes

Gridsize	Grid points	CPU Threads	Inert Uninformative		Myopic		Optimal	
			CPU Time	Memory Usage	CPU Time	Memory Usage	CPU Time	Memory Usage
8x8x8	512	1	0.10	15M	0.09	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

Scaling of CPU-based Computation

- Scaling with CPU threads is sub-linear



GPU-based Computation: High-level Overview

Same approach as for CPU except

- Based on C, not Fortran
- Nested loop over gridpoints to evaluate RHS of Bellman equation is replaced by a call to *kernel* function

From OpenMP to CUDA

F90 Code

```
...
 ! set multithreaded OpenMP version using all available CPUs
#ifdef _OPENMP
  call OMP_SET_NUM_THREADS(numthreads)
#endif
  allocate(V(NX,Nmu,NSigma,2),U(NX,Nmu,NSigma))
...
  do while((ip<MaxPolIter+1).and.(ppass.eq.0))
   ! loop over the grid of the three state variables
   !$omp parallel default(none) &
   !$omp shared(NX,Nmu,NSigma,U,V,X,mu,Sigma,alpha,gamma,delta,omega,ustar,xstar,sigmasq_epsilon) &
   !$omp private(i,j,k)
   !$omp do
   do i=1,NX
      do j=1,Nmu
         do k=1,NSigma
            V(i,j,k,2) = F(U(i,j,k),i,j,k,V)
         enddo
      enddo
   enddo
   !$omp end do
   !$omp end parallel
...
  enddo
...
```

From OpenMP to CUDA

CUDA Code

```
...
    cudaMalloc((void**) &d_X,NX*sizeof(double));
    cudaMemcpy(d_X,X,NX*sizeof(double),cudaMemcpyHostToDevice);
...
    numBlocks=512;
    numThreadsPerBlock=180;
    dim3 dimGrid(numBlocks);
    dim3 dimBlock(numThreadsPerBlock);
...
    while ((ip<MaxPolIter+1)&&(ppass==0))
    {
       // update expected cost-to-go function on the whole grid (in parallel)
       UpdateExpectedCTG_kernel<<<dimGrid,dimBlock>>>(d_U,d_X,d_mu,d_Sigma,d_V0,d_rno,d_wei,d_V1);
       cudaThreadSynchronize();

       // move the data from device to host to do convergence checks
       cudaMemcpy(V1,d_V1,NX*Nmu*NSigma*sizeof(double),cudaMemcpyDeviceToHost);
...
       // update value function, directly on the device
       cudaMemcpy(d_V0,d_V1,NX*Nmu*NSigma*sizeof(double),cudaMemcpyDeviceToDevice);
       // update value function on host as well
       cudaMemcpy(V0,V1,NX*Nmu*NSigma*sizeof(double),cudaMemcpyHostToHost);
...
    }
...
    cutStopTimer(timer);
    gputime = cutGetTimerValue(timer);
    printf("Elapsed GPU Time %5.7f ms.",gputime);
    cudaFree(d_X);
...
```

From OpenMP to CUDA

CUDA Kernel Code

```
...
__device__ inline double UpdateExpectedCTG(double u, double x, double mu, double Sigma,
     double alpha, double gamma, double delta, double omega, double sigmasq_epsilon,
     double xstar, double ustar, int NX,int Nmu, int NSigma, int NGH, double* XGrid,
     double* muGrid, double* SigmaGrid, double* V, double* rno, double* wei);
__global__ void UpdateExpectedCTG_kernel(double* U,double* X,double* mu,
     double* Sigma,double* V0,double* rno,double* wei,double* V1)
{
    //Thread index
    const int      tid = blockDim.x * blockIdx.x + threadIdx.x;
    const int NUM_ITERATION= dc_NX*dc_Nmu*dc_NSigma;
    int ix,jmu,kSigma;

    //Total number of threads in execution grid
    const int THREAD_N = blockDim.x * gridDim.x;

    //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);
    }

}
...
```

Speed Comparisons

- Double precision should be 8 times slower but it isn't
- Single precision requires 10%-50% more iterations to convergence, especially for finer grids

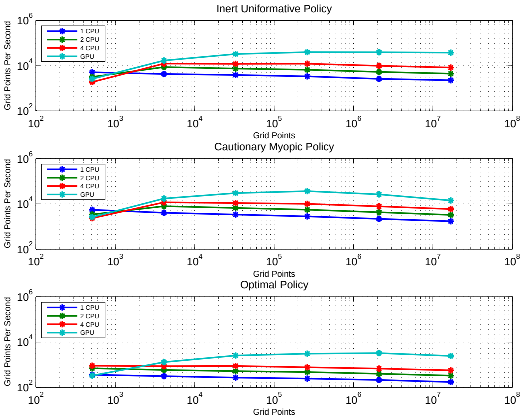
Policy	**Gridsize**	**Grid Points**	**Single Precision Timings**				
			CPU_1	CPU_4	GPU	$\frac{\text{CPU}_1}{\text{GPU}}$	$\frac{\text{CPU}_4}{\text{GPU}}$
Inert Uninformative Policy	8x8x8	512	0.114	0.723	0.176	0.65	4.11
	16x16x16	4,096	0.735	0.689	0.216	3.40	3.18
	32x32x32	32,768	7.039	2.669	0.345	20.40	7.74
	64x64x64	262,144	74.250	25.319	4.598	16.15	5.51
	128x128x128	2,097,152	748.119	223.696	38.197	19.59	5.86
	256x256x256	16,777,216	6,400.123	1,950.315	314.495	20.35	6.20
Cautionary Myopic Policy	8x8x8	512	0.09	0.521	0.172	0.52	3.02
	16x16x16	4,096	1.100	0.806	0.222	4.95	3.63
	32x32x32	32,768	11.397	4.056	0.869	13.12	4.67
	64x64x64	262,144	124.663	40.941	6.935	17.98	5.90
	128x128x128	2,097,152	1,218.964	383.088	78.809	16.36	4.87
	256x256x256	16,777,216	13,739.599	4,161.66	1,494.84	9.19	2.78
Optimal Policy	8x8x8	512	1.639	0.633	1.181	1.39	0.54
	16x16x16	4,096	16.105	5.287	1.764	9.13	3.00
	32x32x32	32,768	153.816	48.754	9.357	16.44	5.21
	64x64x64	262,144	1,413.794	422.316	69.109	20.46	6.11
	128x128x128	2,097,152	16,783.030	5,200.646	829.216	20.24	6.27
	256x256x256	16,777,216	198,708.900	61,810.338	13,466.169	14.76	4.59

Double Precision

Policy	Gridsize	Grid Points	Double Precision Timings				
			CPU_1	CPU_4	GPU	$\frac{\text{CPU}_1}{\text{GPU}}$	$\frac{\text{CPU}_4}{\text{GPU}}$
Inert Uninformative Policy	8x8x8	512	0.096	0.27	0.195	0.51	1.38
	16x16x16	4,096	0.95	0.33	0.295	3.22	1.12
	32x32x32	32,768	8.42	2.71	1.058	7.96	2.56
	64x64x64	262,144	77.17	21.73	7.554	10.22	2.88
	128x128x128	2,097,152	798.45	211.18	61.159	13.06	3.45
	256x256x256	16,777,216	7368.56	2016.57	511.300	14.41	3.94
Cautionary Myopic Policy	8x8x8	512	0.094	0.22	0.197	0.48	1.12
	16x16x16	4,096	1.01	0.35	0.309	3.27	1.13
	32x32x32	32,768	9.72	3.01	1.094	8.88	2.75
	64x64x64	262,144	94.07	26.14	8.12	11.59	3.22
	128x128x128	2,097,152	962.46	270.37	90.453	10.64	2.99
	256x256x256	16,777,216	9,880.06	2,855.14	1,280.381	7.72	2.23
Optimal Policy	8x8x8	512	1.43	0.57	1.564	0.91	0.36
	16x16x16	4,096	13.22	4.79	3.180	4.16	1.51
	32x32x32	32,768	122.68	37.56	12.959	9.47	2.90
	64x64x64	262,144	1,085.47	344.43	85.683	12.67	4.02
	128x128x128	2,097,152	9,972.39	3,131.724	648.598	15.38	4.83
	256x256x256	16,777,216	98,253.32	29,972.30	6,930.597	14.18	4.32

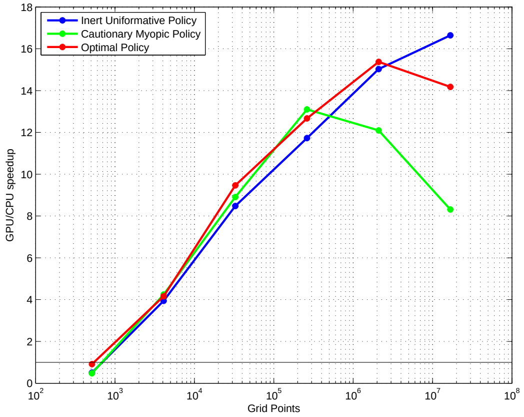
Performance Scaling

- Thread creation/destructon outweighs performance for small problems
- Memory is the limiting factor for all approaches



GPU versus CPU

- Achieving 20x speedup with CPU would be much costlier
- GPU is only worth the effort for moderately large problems



GPU Performance Profiling

Limiting factors

- Limited by registers per microprocessor (low occupancy)
- Optimize occupancy via choice of blocksizes
- Memory bandwidth

Policy	Computation	GPU Time	Average Occupancy	Memory Transfer Size	Registers per Thread	Non-coherent Global Memory Loads	Divergent Branches
Inert Policy	GPU⟶CPU memory copy	0.148		2,097,152			
	Kernel Execution	5.400	18.8%		71		0.75%
	GPU⟶GPU memory copy	0.000					
	CPU⟶GPU memory copy	0.074		2,097,152			
Cautionary Myopic Policy	GPU⟶CPU memory copy	0.148		2,097,152			
	Kernel Execution	5.950	18.8%		71		5.21%
	GPU⟶GPU memory copy	0.000					
	CPU⟶GPU memory copy	0.074		2,097,152			
Optimal Policy	GPU⟶CPU memory copy	0.050		2,097,152			
	Kernel Execution	84.260	12.5%		122		3.83%
	GPU⟶GPU memory copy	0.007					
	CPU⟶GPU memory copy	0.001		2,097,152			

Odds and Ends

Performance left on the table

- Non-specific
  - Asynchronous Gauss-Seidel sweeps can speed convergence and reduce memory usage
  - Can combine value iterations with policy evaluation iterations
- CUDA-specific
  - Norm comparison is on CPU and is inefficient
  - Tune register use using compiler switch
- Advanced CUDA
  - Multilinear interpolation can use textures
  - Pinned memory to overlap computation and communication
  - Atomic functions to reduce memory scattering

- GUI is less responsive during runs unless multiple cards or Tesla cards used

Conclusions

We learned

- Low hanging fruit is still available
- CUDA is easy to pick up but hard to push to the limit
- Must rethink your algorithms to be aggressively parallel
  - not just a good idea, but the only way
  - otherwise, if it is not fast enough, it will never be

Future trends

- Computers no longer get faster, just wider
- Easier and more flexible programming tools
  - PGI compilers with support for accelerator directives coming soon
- Heterogeneous computing
- Competing standards, hardware and tools