GPU Computing in Economics

Eric M. Aldrich

UC Santa Cruz

Overview

New developments in computer architecture and hardware allow scientists to solve problems that were previously infeasible.

- I will introduce you to resources that allow for massively parallel computation and demonstrate their relevance to economic problems.
- We will focus on graphics processing units (GPUs).

Graphics Processing Units

The gaming industry is largely responsible for the development of a new multi-core technology for rendering graphics.

- The huge demand for complex, real-time graphics in computer games, led to powerful GPUs.
  - GPUs now have hundreds or thousands of processors.
  - They were originally designed to handle millions of identical floating-point operations (rendering pixels).
- In the mid 2000s, scientists began leveraging graphics processors for highly parallel algorithms.
- Accessing graphics processors was initially very difficult.
- This improved with the release of the NVIDIA CUDA toolkit in 2007.

GPU Processors

GPUs are comprised of dozens to thousands of individual processing cores.

- These cores, known as thread processors, are typically grouped together into several distinct multiprocessors.
- The Tesla C2075 has a total of 448 thread processors, aggregated into groups of 32 cores per multiprocessor, yielding a total of 14 multiprocessors

GPU Processors

Relative to CPUs, GPU cores typically:

- Have a lower clock speed. Each Tesla C2075 core clocks in at 1.15 GHz.
- Dedicate more transistors to arithmetic operations and fewer to control flow and data caching.
- Have access to less memory. A Tesla C2075 has 6 gigabytes of global memory, shared among all cores.

GPU Processors

Where GPU processors are lacking in clock speed and memory access, they compensate with sheer quantity of compute cores.

- They are ideal for computational work that has a high arithmetic intensity: many arithmetic operations for each byte of memory transfer/access.
- The broad concept is that the GPU is ideal for simple arithmetic instructions that require little memory access and no branching.

GPU Processors

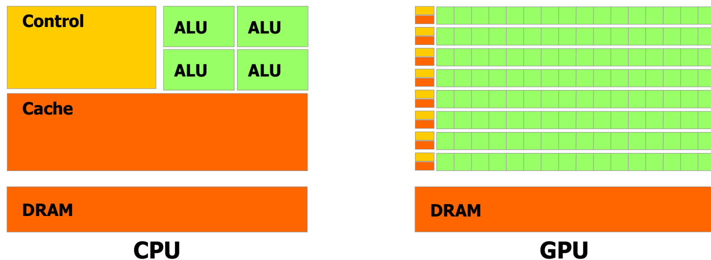


Figure 1 : Figure 3 from the NVIDIA CUDA C Programming Guide, Version 4.2

GPU Memory

There is a distinction between CPU memory and GPU memory.

- CPU memory is referred to as ‘host’ memory.
- GPU memory is referred to as ‘device’ memory.
- GPU instructions can only operate on data objects in device memory.
- GPU software design typically necessitates the transfer of data objects between host and device memory.

Scalability

Two notions of scalability are relevant to GPU computing: scaling within GPU devices and across GPU devices.

- GPUs automatically handle within-device scaling when software is moved to GPU devices with differing numbers of thread processors.
- The scheduler deals with scalability so that issues related to processor count and processor interaction on a specific device are transparent to the user.
- This increases the portability of massively parallel GPU software.
- Scalability across GPU devices is achieved in a more traditional manner, using MPI.

Amdahl's Law

For a given computational program, there is no guarantee that any portion of the program may be computed in parallel.

- Most algorithms have some fraction of instructions which must be performed sequentially, and a remaining fraction which may be implemented in parallel.
- Amdahl's Law states that if a fraction P of a program can be executed in parallel, the theoretical maximum speedup of the program with N processors is

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}.$$

- Intuition: if a serial version of an algorithm takes 1 unit of time to execute, a fraction $(1-P)$ of a parallel algorithm will execute in the same time as its serial counterpart, whereas a fraction P will run in P/N units of time (on N processors).

Kernels and Threads

Kernels and threads are the fundamental elements of GPU computing problems.

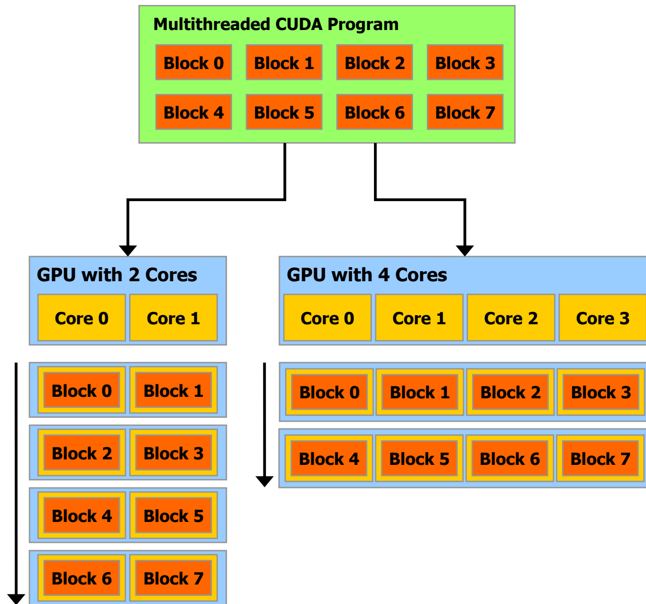
- Kernels are special functions that comprise a sequence of instructions that are issued in parallel over a user specified data structure.
- Thus, a Kernel typically comprises only a portion of the total set of instructions within an algorithm.
- Each data element and corresponding kernel comprise a thread, which is an independent problem that is assigned to one GPU core.

Kernels and Threads

Just as GPU cores are grouped together as multiprocessors, threads are grouped together in user-defined groups known as blocks.

- Thread blocks execute on exactly one multiprocessor.
- Typically many thread blocks are simultaneously assigned to the same multiprocessor.
- The scheduler on the multiprocessor then divides the user defined blocks into smaller groups of threads that correspond to the number of cores on the multiprocessor.
- Each core of the multiprocessor then operates on a single thread, issuing each of the kernel instructions in parallel.
- This architecture is known as Single-Instruction, Multiple-Thread (SIMT).

Threads and Thread Blocks



`CUDA C`

NVIDIA was the original leader in developing a set of software tools allowing scientists to access GPUs.

- The `CUDA C` language is a set of functions that can be called within basic C/C++ code that allow users to interact with GPU memory and processors.
- `CUDA C` is currently the most efficient and best documented way to design GPU software.
- Downsides: it requires low-level comfort with software design (similar to C/C++) and that it only runs on NVIDIA GPUs running the CUDA platform.
- The CUDA platform itself is free, but requires NVIDIA hardware.

OpenCL

OpenCL is an open source initiative lead by Apple and promoted by the Khronos Group.

- The syntax of OpenCL is very similar to CUDA C.
- It has the advantage of not being hardware dependent.
- It is intended to exploit the heterogeneous processing resources of differing GPUs and CPUs simultaneously within one system.
- Downsides: it is poorly documented and has much less community support than `CUDA C`.

Other Tools

- Matlab Parallel Computing Toolbox.
- Mathematica CUDALink and OpenCLLink.
- PyCUDA.
- AccelerEyes Jacket (for Matlab) and ArrayFire (for C, C++ and Fortran).
- Thrust, ViennaCL and C++AMP.

Heterogeneous Beliefs Example

Consider an economy with the following elements:

- Discrete time, indexed by $t \in \mathcal{N}_0 = \{0, 1, 2, \ldots\}$.
- Aggregate states $s_t \in \mathcal{S} = \{1, \ldots, S\}$ at each t, where we denote the history of states as $s^t = (s_0, s_1, \ldots, s_t)$.
- I agent types, indexed by $i \in \mathcal{I} = \{1, \ldots, I\}$.
- Agent type proportions $\mu^i(s^t)$ such that the total population has unit mass, $\sum_{i=1}^{I} \mu^i(s^t) = 1$, for all $s^t \in \mathcal{S}^t$.

Endowments and Preferences

There is a single consumption good and a tree paying dividend $d(s^t)$ units of the consumption good in state s^t.

- Agents are endowed with $d(s^t)$ in each state, resulting in an endowment of $\mu^i(s^t)d(s^t)$ for cohort i and an aggregate endowment of $\sum_{i=1}^{I} \mu^i(s^t)d(s^t) = d(s^t)$.
- Agents have (potentially) heterogeneous period utility functions over consumption, $u_i(c)$, which satisfy the usual conditions of strict monotonicity, strict concavity, twice continuous differentiability and $\lim_{c\to 0} u'_h(c) = \infty$.

Beliefs

The aggregate state follows an S-state Markov process.

- $\pi(s^t) = \pi(s_t|s_{t-1}) \cdots \pi(s_1|s_0)\pi(s_0)$, where $s_0 \in \mathcal{S}$ is known and hence $\pi(s_0) = 1$.

Agents have heterogeneous beliefs about the transition probabilities.

- $\pi^i(s^t) = \pi^i(s_t|s_{t-1}) \cdots \pi^i(s_1|s_0)$.

Optimization Problem

Allowing agents to purchase state-contingent consumption $c^i(s^t)$ at prices $q^0(s^t)$, each agent solves:

$$U_i(\boldsymbol{c}^i) = \max_{c^i} \left\{ u(c^i(s_0)) + \sum_{t=1}^{\infty} \beta_i^t \sum_{s^t} u_i(c^i(s^t))\pi^i(s^t) \right\} \tag{1a}$$

subject to

$$c^i(s_0) + \sum_{t=1}^{\infty} \sum_{s^t} q^0(s^t) c^i(s^t) \leq d(s_0) + \sum_{t=1}^{\infty} \sum_{s^t} q^0(s^t) d(s^t), \tag{1b}$$

where $\boldsymbol{c}^i = (c(s_0), c(s^1), \ldots)$ and where $q^0(s_0) = 1$.

Competitive Equilibrium

A competitive equilibrium for this economy is a collection of consumption plans $\{\bar{\boldsymbol{c}}^i\}_{i=1}^I$ and prices $\left\{\{\bar{q}^0(s^t)\}_{s^t \in \mathcal{S}^t}\right\}_{t=0}^{\infty}$ such that

1. System (1) is solved.
2. The aggregate resource constraint

$$\sum_{i=1}^{I} \mu^i(s^t)\bar{c}^i(s^t) = \sum_{i=1}^{I} \mu^i(s^t) d(s^t) \tag{2}$$

   holds for all $s^t \in \mathcal{S}^t$ and $t \geq 0$.

Euler Equation

The intertemporal Euler equation for each agent is

$$\beta_i^t \frac{u_i'(c^i(s^t))}{u_i'(c^i(s_0))} \pi^i(s^t) = q^0(s^t), \tag{3}$$

for all $s^t \in \mathcal{S}^t$ and $i = 1, \ldots, I$. Selecting agent 1 as a "reference" agent, Equation (3) yields

$$\frac{\beta_i^t}{\beta_1^t} \frac{u_i'(c^i(s^t))/u_i'(c^i(s_0))}{u_1'(c^1(s^t))/u_1'(c^1(s_0))} \frac{\pi^i(s^t)}{\pi^1(s^t)} = 1. \tag{4}$$

Optimal Consumption

Reformulating (4), we arrive at

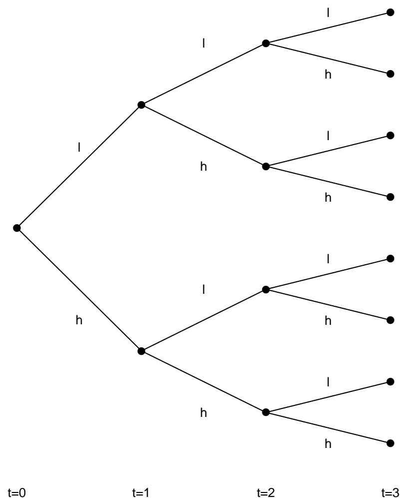
$$c^i(s^t) = u_i'^{-1}\left(\frac{\beta_1^t}{\beta_i^t}\frac{\pi^1(s^t)}{\pi^i(s^t)}\frac{u_1'(c^1(s^t))}{u_1'(c^1(s_0))}u_i'(c^i(s_0))\right). \tag{5}$$

Substituting (5) into the aggregate resource constraint (2),

$$\sum_{i=1}^{I}\mu^i(s^t)u_i'^{-1}\left(\frac{\beta_1^t}{\beta_i^t}\frac{\pi^1(s^t)}{\pi^i(s^t)}\frac{u_1'(c^1(s^t))}{u_1'(c^1(s_0))}u_i'(c^i(s_0))\right) = \sum_{i=1}^{I}\mu^i(s^t)d^i(s^t). \tag{6}$$

Given $\{c^i(s_0)\}_{i=1}^{I}$, and the other economic fundamentals, Equation (6) represents a single nonlinear equation with a single unknown, $c^1(s^t)$.

Example of a State Tree



Finite-Period Algorithm

We can solve a finite-period version of the model with the following algorithm.

1: Fix some $\tau > 0$, which will determine convergence and set $\varepsilon = 1$.
2: Guess initial values for $c^i(s_0)$, $i = 2, \ldots, N$.

Finite-Period Algorithm

3: for $s^t \in \mathcal{S}^t \setminus s^0$

4: Determine $c^1(s^t)$ such that

$$\sum_{i=1}^{I} \mu^i(s^t) u_i'^{-1} \left(\frac{\beta_1^t}{\beta_i^t} \frac{\pi^1(s^t)}{\pi^i(s^t)} \frac{u_1'(c^1(s^t))}{u_1'(c^1(s_0))} u_i'(c^i(s_0)) \right) = \sum_{i=1}^{I} \mu^i(s^t) d(s^t). \quad (7)$$

5: for $i = 2, \ldots, I$

6: Compute

$$q^0(s^t) = \beta_1^t \frac{u_1'(c^1(s^t))}{u_1'(c^1(s_0))} \pi^1(s^t) \quad (8)$$

$$c^i(s^t) = u_i'^{-1} \left(\frac{\beta_1^t}{\beta_i^t} \frac{\pi^1(s^t)}{\pi^i(s^t)} \frac{u_1'(c^1(s^t))}{u_1'(c^1(s_0))} u_i'(c^i(s_0)) \right). \quad (9)$$

7: end for

8: end for

Finite-Period Algorithm

9: for $i = 2, \ldots, N$

10: Compute

$$\varepsilon^i = d(s_0) + \sum_{t=1}^{T} \sum_{s^t} q^0(s^t) d(s^t) - c^i(s_0) - \sum_{t=1}^{T} \sum_{s^t} q^0(s^t) c^i(s^t). \tag{10}$$

11: end for

12: Set $\varepsilon = \sum_{i=2}^{N} |\varepsilon^i|$.

13: if $\varepsilon > \tau$ then

14: Use Broyden's method to choose new values of $c^i(s_0)$ and return to Step 3.

15: else

16: Stop.

17: end if

Parallel Computing

The algorithm above is well suited for parallel computing.

- The independence between states allows the computation to be divided into distinct pieces, each of which can be performed by a separate processing unit.
- In theory, with enough processors, it would be possible to assign the nonlinear equation problem of each state to one processor.
- In practice, with large S or large T, a subset of state tree nodes would be assigned to each processor for computation.

Results

T	5	10	15	20	22	24	26	28	30
Thrust/OpenMP	0.0004110	0.002138	0.04968	1.1934	6.864	35.88	113.0	461.1	7021
Thrust/CUDA	6.630	6.646	6.606	6.686	6.767	7.507	10.15	23.40	67.15

Conclusions

- GPUs are specialized devices that can greatly speed computation through massive parallelism.
- Economic problems that traditionally would have taken hours can now be performed in seconds.
- This allows researchers to achieve greater precision in their work or explore state spaces or models that were previous intractable.
- The gains from GPU computing depend on the nature of the algorithm at hand and the portion that can be computed in parallel - GPU computing is not a panacea.
- With some creativity, many seemingly serial problems can be cast in a parallel fashion.