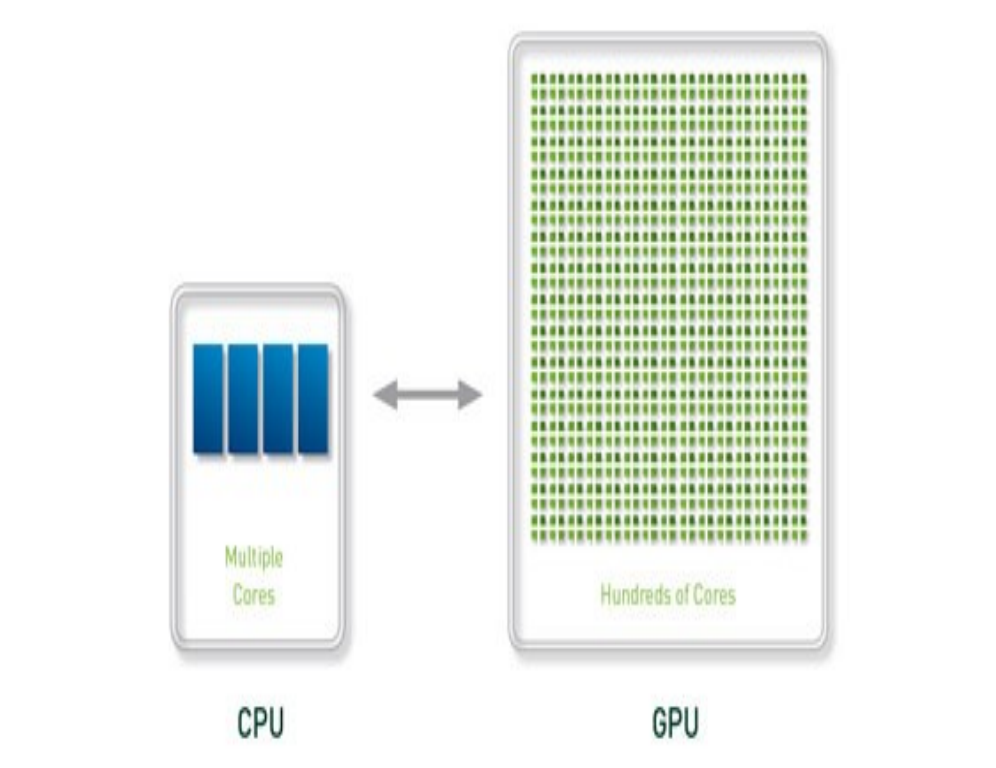
Proper Optimization Tactics and Techniques for Full Scale Performance Realizations on Graphics Processing Units

By Justin McKennon

Why choose the GPU?

- If you've ever played any computer games, you have already seen the power of the GPU.
- Explosions, in-game physics, image rendering – all of these calculations are done by the GPU at an extremely fast rate.
- It wasn't until recently that the scientific community has become aware of the potential that GPU computing has in scientific applications
- Several computing languages have been developed to allow users to harness the raw supercomputing power that lives inside the GPU
- CUDA, OpenCL, CUDAFortran and many more

Why choose the GPU?



- The most powerful PC's in the world have 8 cores.
- Most GPUs are filled with over 120 cores
- Tesla C1060 has 240 cores that run at 1.3 GHz
- NVIDIA Fermi has 512 cores
- The power of GPUs has been known since the advent of graphics

If GPUs are so powerful, why isn't everything run solely on them? Why doesn't everyone use them?

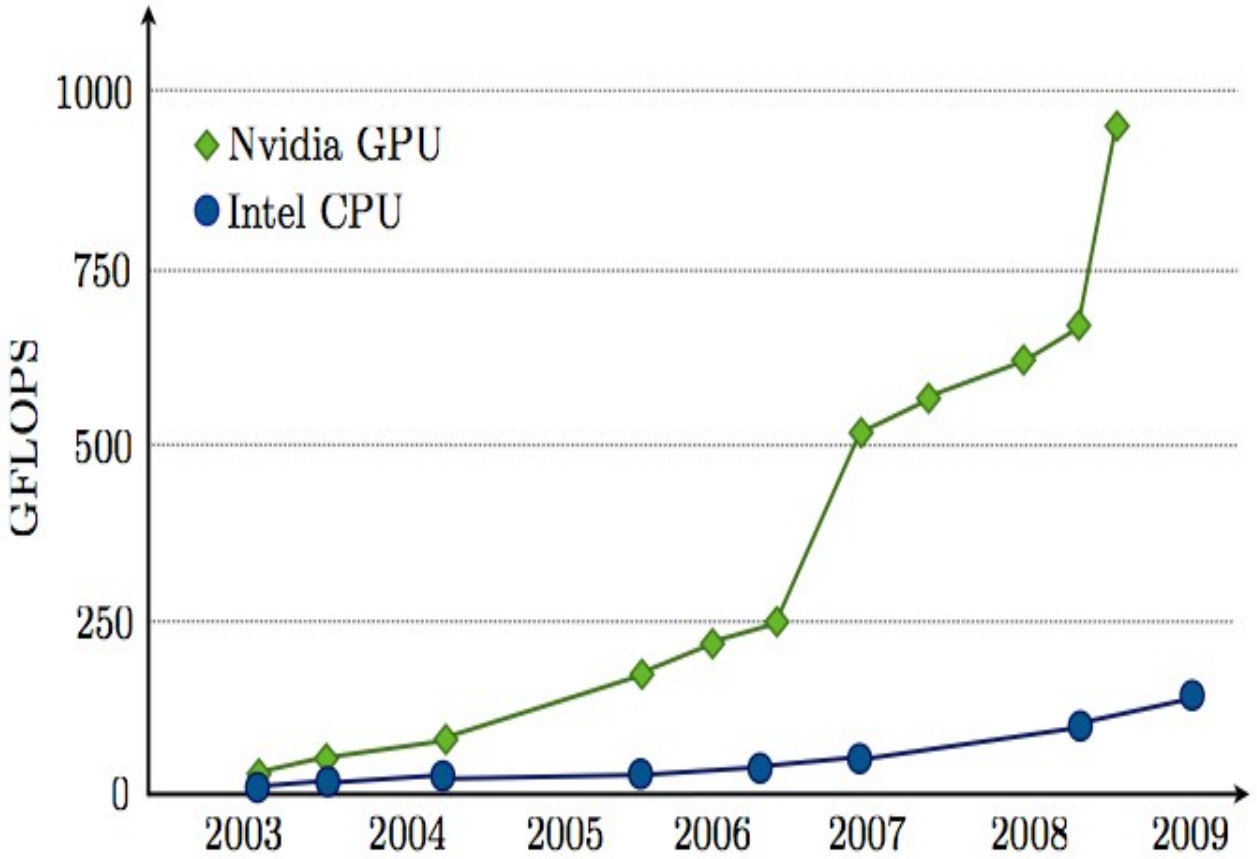
- Serial CPU computing is easy.
- There are many experts and a variety of programming languages (C, Java, Python, FORTRAN etc)
- Utilizing the GPU is much more difficult : no operating system, complex memory management, very few programming language options, difficult to set up, costly data transfer between host and device
- Dozens of cons to using the GPU – but for applications where using the CPU as the main computational component (scientific computing) is simply impossible, the GPU excels

The GPU Devotes More Transistors to Data Processing

The motivation

GPU computing - key ideas:

- Massively parallel.
- Hundreds of cores.
- Thousands of threads.
- Cheap.
- Highly available.
- Programable: CUDA



Unlocking the power of GPUs

- In recent years, several GPU specific programming languages have been developed.
- The most well known and widely used language was developed by NVIDIA and is called CUDA.
- CUDA stands for Compute Unified Device Architecture
- CUDA is a C derivative and has many of the same programming standards as C, with some restrictions.

- So now you have your code written in CUDA and can run it on the GPU. What next?
- Most GPU programmers call it a day.
- But now, with the advances in the science of GPU based computing we are now able to extract substantial performance gains by refining GPU code.
- These refinements are done by examining and tip-toeing around the bottlenecks of GPU based computing: memory and thread management.

Optimizing Your Code

Code Optimization

- The first step that needs to be taken when optimizing GPU code is profiling.
- Languages such as CUDA come with a profiler that analyzes the code and tell you exactly what it is doing.
- Through the profiler, you can learn an awful lot about how your code is performing.

Code Optimization

- When examining the results of the profiler, there are several things that need to be looked for.
- Divergent branches, local loads and stores, warp serializations and incoherent loads and stores.
- When any of these occur in GPU code, the performance of the code is being robbed, in most cases by up to 70% of the possible performance gain.

Divergent Branches

- Thread divergence is determined on a half warp level. This is due to the fact that threads of a half warp execute the same instructions.
- On newer devices, a warp is 32 threads. This means that a half warp is 16 threads.
- Divergent branches are not to be confused with regular branches.

Divergent Branches

- In any sort of useful code, avoiding branches all together is impossible.
- If statements, switch statements – things that have multiple outcomes, all have branches.
- If all the threads in a half warp enter the same branch there is no performance lost.
- A=2;
- If(A>1)
  {some code}
- The profiler will report a branch even though all the threads will take the same path here.

Divergent Branches

- The problem arises when threads of the same half warp are forced to execute different instructions. When this occurs, the instructions are forced to be serialized. Each set of threads in the divergent branch takes turns executing their part of the branch and then resume together outside the branch once each of the threads has had it's turn.
- Divergent branches occur most often in code that manipulates threads directly. By scheduling threads manually you can unintentionally force instructions to be serialized. To avoid this, it is best to use parameters not based on thread indices when dealing with branches.

Local Loads and Stores

- When dealing with automated memory management, the GPU defaults to using registers (declaring variables etc.)
- Registers are small storage containers that provide the fastest form of data access in any system.
- Local loads and stores occur when there are simply not enough registers to hold the data and it spills over into local memory.
- 

Ice Cube Tray Example

- The registers on a GPU are similar to an ice cube tray.
- As water (data) fills up the sections of an ice cube tray, each section will become full with a certain amount of water.
- When the tray is full and more water is continuing to be added, the tray will eventually overflow. This is exactly what happens with local loading and storing.
- When the registers overflow, the GPU then “cleans up” the mess from the spillage and reallocates everything into local memory. This whole process is very time consuming and to top it off, local memory is 8x slower than register based data accessing.
- The problem is that local memory resides in the global memory and can be up to 150x slower than register based or shared memory.

Local Loads and Stores

- Local loads and stores commonly occur when large arrays or pieces of data are allocated on the GPU – be it dynamically or when data is passed by reference to the kernel.
- It is often very difficult to deal with local loads and stores (you have a really big array... what are you supposed to do?)
- Often times when local loads and stores occur, the spilling is very small. By blocking off code into smaller kernels (keeping in mind that kernels that are too small are equally as bad as local loads and stores) you can effectively eliminate the data spillage all together.

Incoherent Loads and Stores

- First, a coherent (or coalesced) memory transaction is one in which all of the threads in a half warp access global memory at the same time. To simplify, the idea is to have all of the load/store operations combined in to a single request by the hardware – e.g. all the threads perform accesses (reads or writes) to memory simultaneously.
- This may be a bit much to digest, but these types of things are some of the easiest and most important things to fix in your code if you want to optimize it completely.
- Having non-coalesced memory transactions means you are doing extra read/write instructions without needing to. If this occurs in a large loop type piece of code, the amount of computation time will dramatically increase.

Incoherent Loads and Stores

- So how do you coalesce your memory transactions?
- Easy:use consecutive memory addresses.
- As variables are created in a program (say for example, the first variable created is located at the address 0x0000), the address increments depending on the type of variable (integer, double, character etc).
- Let's take a look at an example:

Incoherent Loads and Stores

- int a=2; // This variable resides at address 0x0000
  int b=4; // address 0x0004
  int c=6; // 0x0008
  int d=8; // 0x000C
  etc..

- This is what a snapshot of the memory looks like:

  0x0000-->2
  0x0004-->4
  0x0008-->6
  0x000C-->8
  etc

Incoherent Loads and Stores

- So now, when this is processed on the GPU, the threads go to work (let's assume there is a read operation called)

  0x0000-->2 thread1
  0x0004-->4 thread2
  0x0008-->6 thread3
  0x000C-->8 thread4

  etc

- Since each piece of required data falls in consecutive memory locations, the read operation is coalesced and everything can be read in parallel all at once.

Incoherent Loads and Stores

- Now, imagine that this program from the previous example was part of a very large program in which the variables are declared as:

  ```
  int a=2;
  int *b=new int [4];
  int c=6;
  etc..
  ```

  The memory snapshot now looks like:

  0x0000--> 2
  0XC278-->4
  0x0004-->6

Incoherent Loads and Stores

- Notice that since b is a dynamically allocated integer, it does not reside in the same location in memory as the a & c. This means that when the read operation is called, thread 1 will see variable a with no problem. Thread 2 however, will have no idea where b is. The read operation will be stopped and another instruction to go find b's address will be issued, followed by another read operation. Thread 3 now has no idea where c is located. The read operation ends, another instruction to find c's address is issued and then another read operation is performed.
- While this is an extremely trivial example of an incoherent load/store, there are many other ways this kind of thing can occur.

Incoherent Loads and Stores

- These range from easy to spot things such as the previous problem involving obvious address issues, to things such as operations formed on multi-dimensional arrays.
- The key to avoiding these things is to organize your code.
- By organizing your code appropriately you can be certain that you aren't attempting to do something like: reading from an array who has elements that contain pointers to other addresses and things of that nature.

Final Remarks

- When it comes to GPU computing you MUST optimize your code – if you don't than there was no reason to put it on the GPU in the first place!
- Utilize the profiler. It will tell you where your code sucks and where it is awesome.
- When the profiler alerts you to problems with your code, know what is causing it. You can't fix problems if you don't know what they are (like what is a divergent branch?)
- Currently, I am doing all of this for mine and Dr. Khanna's Extreme Mass Ratio Inspiral CUDA simulation code.

References

Mark Harris. Mapping computational concepts to GPUs. In ACM SIGGRAPH 2005 Courses (Los Angeles, California, July 31 – August 4, 2005). J. Fujii, Ed. SIGGRAPH '05. ACM Press, New York, NY, 50.

Felipe A. Cruz, Tutorial on GPU Computing with an Introduction to CUDA, University of Bristol, Bristol, United Kingdom, 2006.

Schatz, M.C., Trapnell, C., Delcher, A.L., Varshney, A. (2007) High-throughput sequence alignment using Graphics Processing Units. BMC Bioinformatics 8:474.

"GPU-Accelerated Time-Domain Circuit Simulation Paper at CICC" . *Signal Integrity*. Agilent Technologies, Inc.. 2 September 2009. Retrieved 3 September 2009.