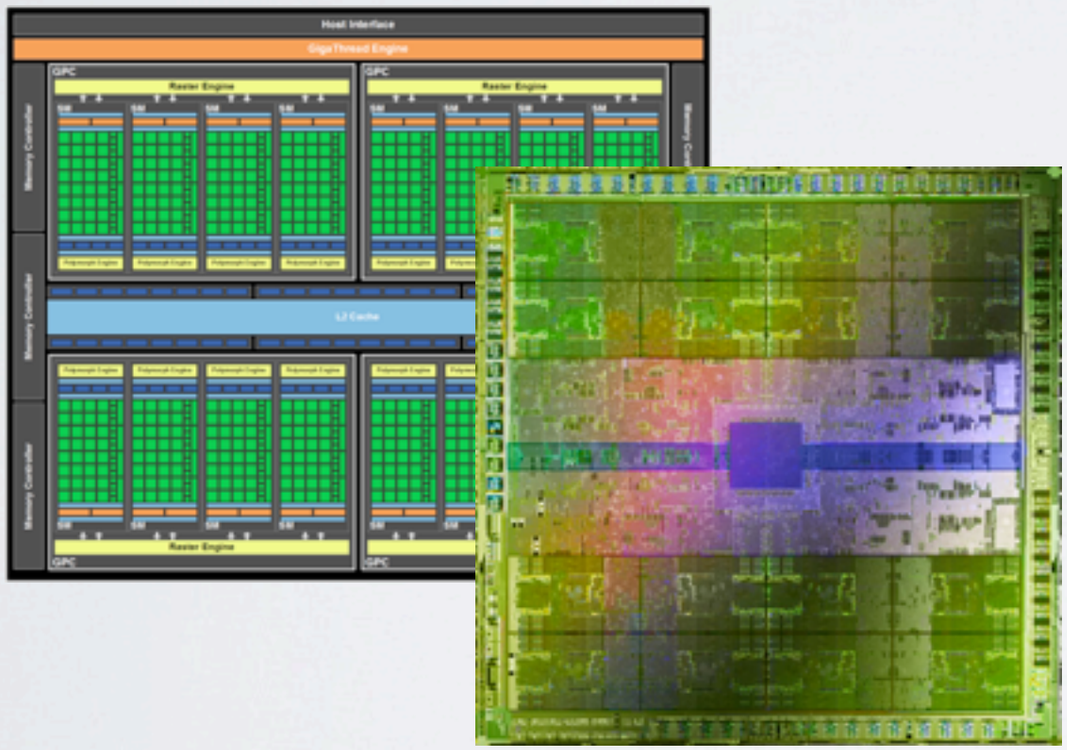
# CUDA
# PARALLEL PROCESSING

By Bill Foland



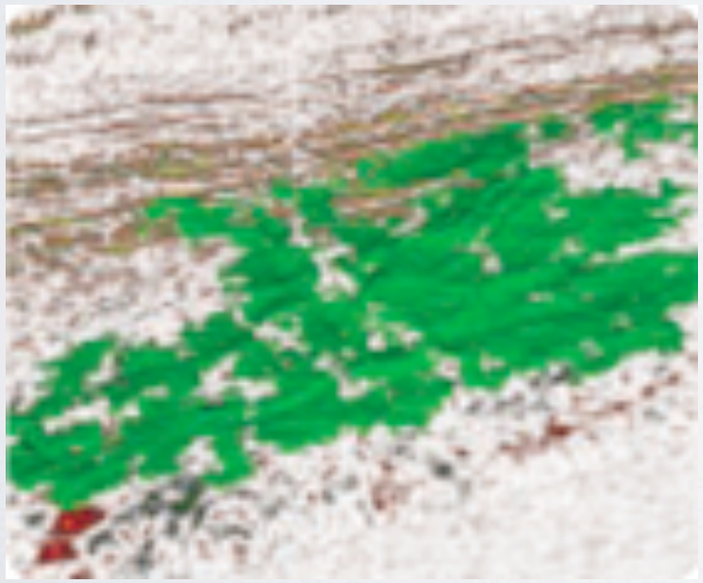
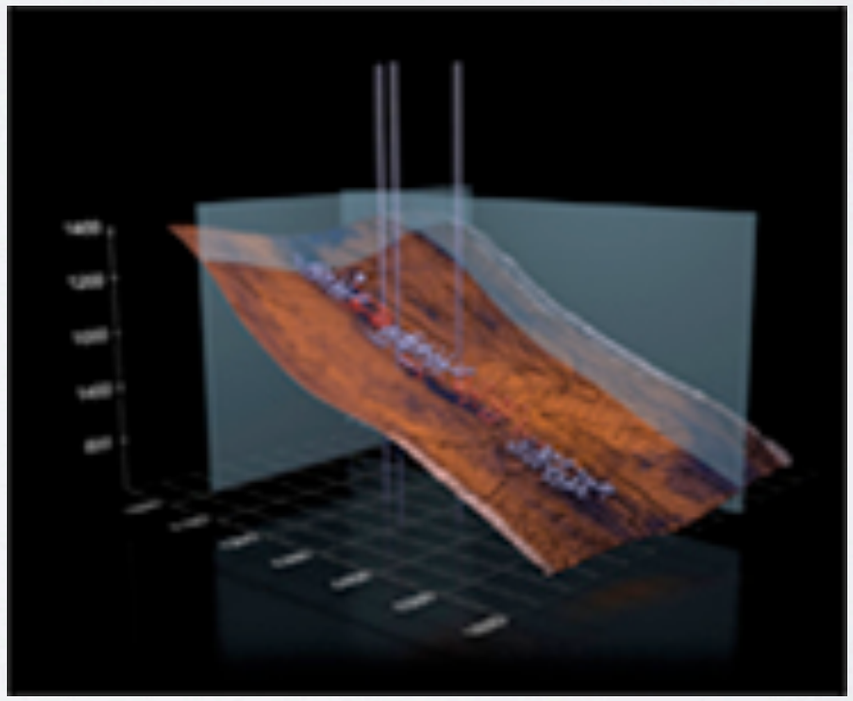

# *WHAT IS CUDA*

- CUDA stands for Compute Unified Device Architecture.
- It's a hardware and software architecture created by NVidia.
- CUDA C gives developers access to the virtual instruction set and memory of Graphics Processing Units (GPUs).
- The approach of solving general purpose problems on GPUs is known as GPGPU.
- GPUs can have hundreds of cores, allowing massively parallel program execution, similar to supercomputing, on the desktop.
- OpenCL and DirectCompute are the competing standards, and NVidia seems to be gradually adding support for OpenCL too.

# *SUPERCOMPUTING*

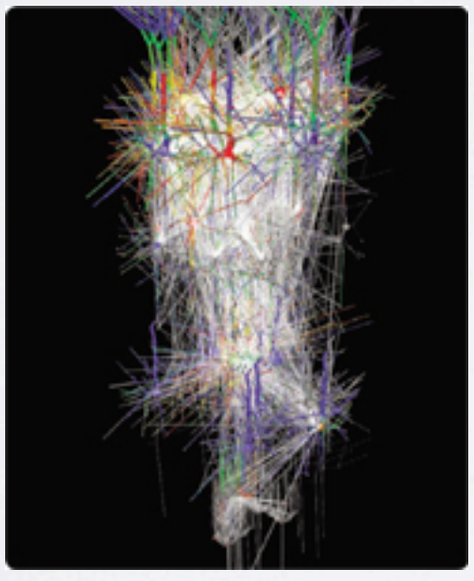
- NVidia packages their high end GPU's for use as a supercomputing module.
- Desktop machines including these modules are sold as desktop supercomputers.
- Clusters of these modules are now included with 35 of the top 500 supercomputing systems
- For example, the Mole-8.5 GPU-accelerated supercomputer, which includes more than 2,200 NVIDIA Tesla GPUs was used recently to simulate and study the whole H1N1 influenza virus. (NVidia Press Release)

# *SEISMIC ANALYSIS*

- The search for oil and gas uses algorithms to analyze seismic data which are computationally intensive.
- Massive numbers of array and FFT calculations, for example.
- A company called GeoStar uses Tesla GPU's, and claims a speedup of 600x over the previously used cluster of 66, 3.4GHz CPU's.
- 


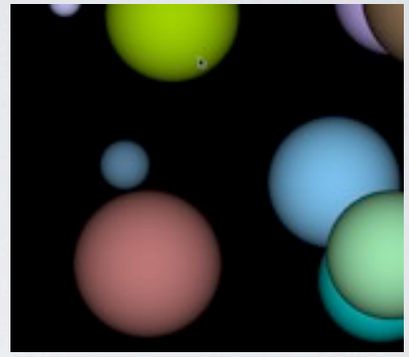
# *NEURAL CIRCUIT SIMULATION*

- Evolved Machines simulates neurobiologically realistic neural circuits which requires gigaflops per neuron.
- A neural array requires thousands of neurons, and so the detailed simulation of neural systems in real time requires >10 teraflops of computing power.
- They claim a 130x performance improvement over a CPU cluster.



# *RAY TRACING*

- Ray tracing is an algorithm used to calculate synthetic images from mathematical descriptions.
- An example of using CUDA to create a simple raytracer spawns one thread per pixel, something never done on a CPU because of the huge overhead. But on a GPU this is practical, since overhead is very low.

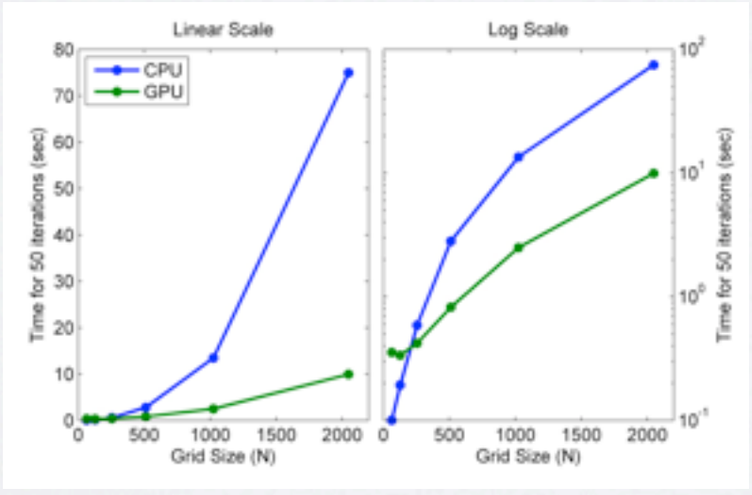


CUDA by Example, Chapter 6

- Weta Digital, in New Zealand, used CUDA to speed up computations for ray tracing by a factor of 25 to create complex scenes for Avatar.

# *MATLAB PARALLEL TOOLBOX*

- Matlab's Parallel Computing Toolbox provides a straightforward way to speed up MATLAB code by executing it on a GPU.
- You simply change the data type of a function's input to take advantage of the many MATLAB commands that have been overloaded for GPUArrays.
- A factor of 10x for some algorithms is claimed.
- A list of accelerated functions supported is here.



# *WILL CUDA SPEED UP MY APP?*

- It must be capable of being broken down into thousands of similar, but independent, units of work for massive parallelism.
- It must be computationally intensive. I/O intensive algorithms won't benefit, for example.
- The intense data parallelism provided by a GPU is tough to take advantage of without some understanding of the hardware. It's a different intuition from CPU multi-core threads.
- More on this later.

# *EARLY HISTORY*

- Early 2000's Microsoft DirectX standard pushed GPU manufacturers to create a more programmable device.
- 2001 GPU's were designed to produce to produce a color for every pixel on the screen using pixel shaders.
- The massive array of ALUs was too hard to resist, so researchers explored general purpose computation through a convoluted process:
  - The GPU was tricked into performing non-rendering tasks by making those tasks appear as if they were a standard rendering.
  - The GPU pixel color computation was manipulated to be any data of interest.

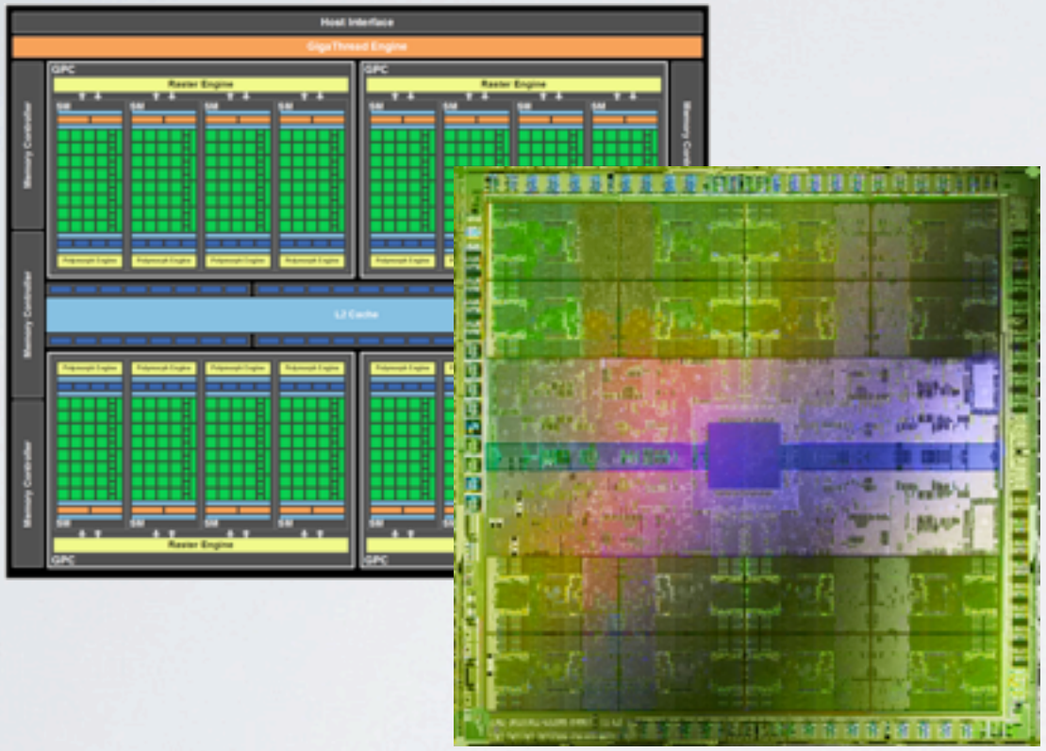
# *EARLY HISTORY CONTINUED*

- Billions of Floating point operations per second were accomplished, but problems with early usage GPGPU included:
  - Using OpenGL and DirectX was clumsy.
  - ALU's were not fully functional.
  - Memory accesses were awkward.
  - Intense understanding of the inner workings of the hardware was required.

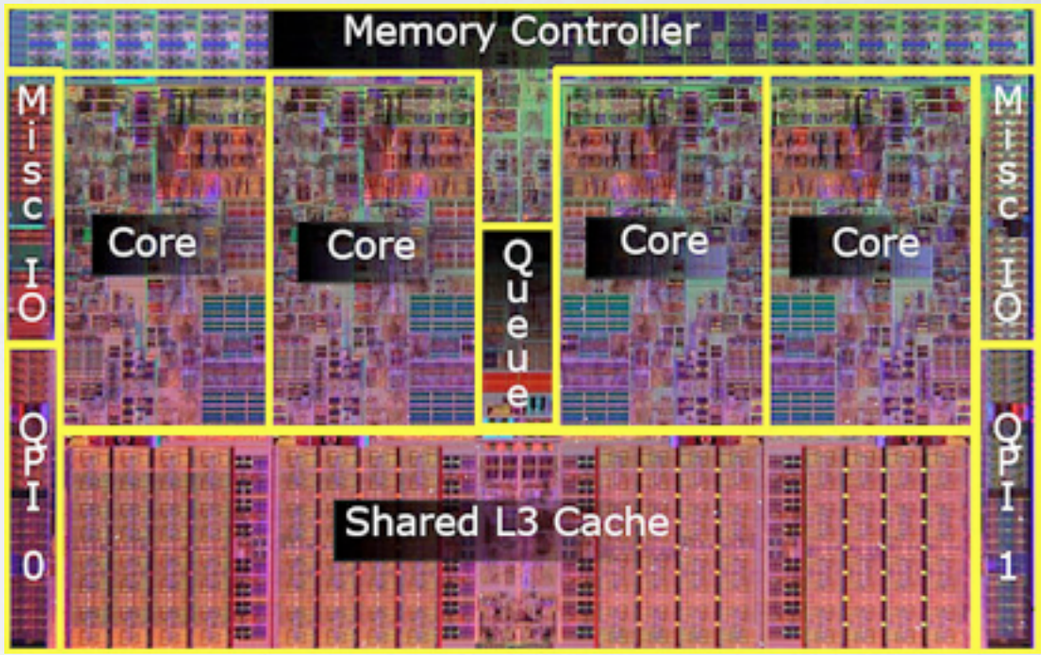
# *CUDA HARDWARE CHANGES*

- CUDA hardware architecture includes new components designed strictly for GPU computing.
  - ALUs improved to IEEE single precision arithmetic support
    - (and double precision later)
  - General computation instruction set
  - Execution units modified to allow arbitrary r/w access to memory.
  - Added a software managed cache for shared memory.

# DIFFERENT ARCHITECTURES




NVidia Fermi

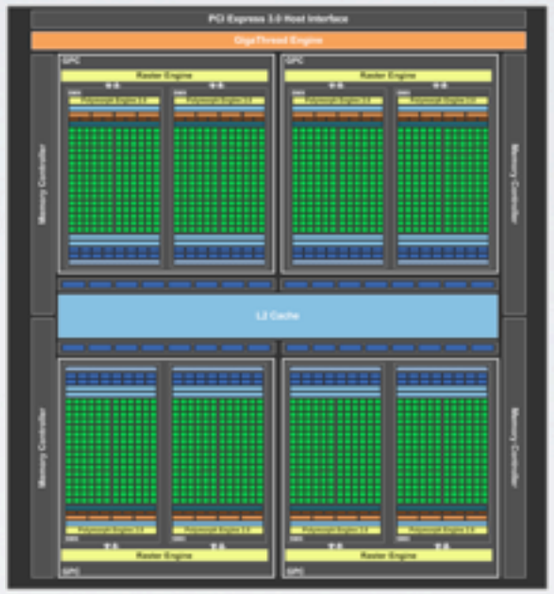


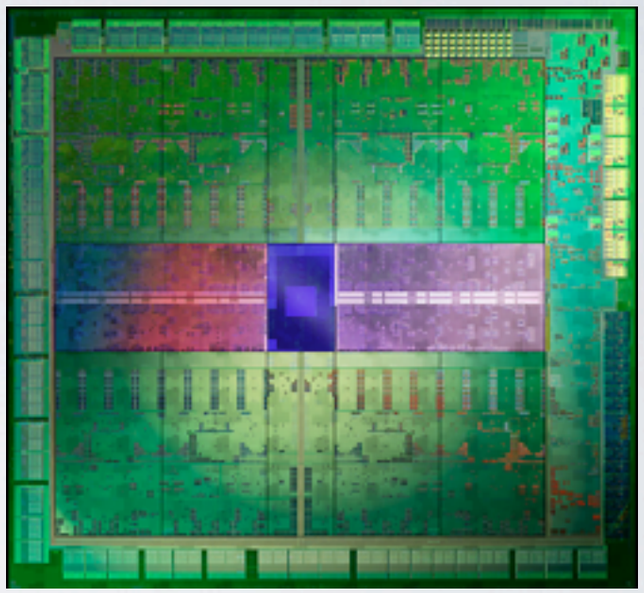

Intel Quad Core i7

- It takes a lot of hardware to implement flexible CPU cores.
- A much larger fraction of the die is used for ALUs in a GPU.

# *AND THE TREND CONTINUES*

- Recently, NVidia released details about their new GPU, Kepler.
- Lower Clock speed,
- Triple the cores to 1536
- 170 Watts Max (Board)



- Twice the performance per Watt vs. Fermi.
- Nvidia Shows Off First 'Kepler' GPUs – PCs first, Server GPU coprocessors in Q3 | insideHPC.com

# *CUDA C*

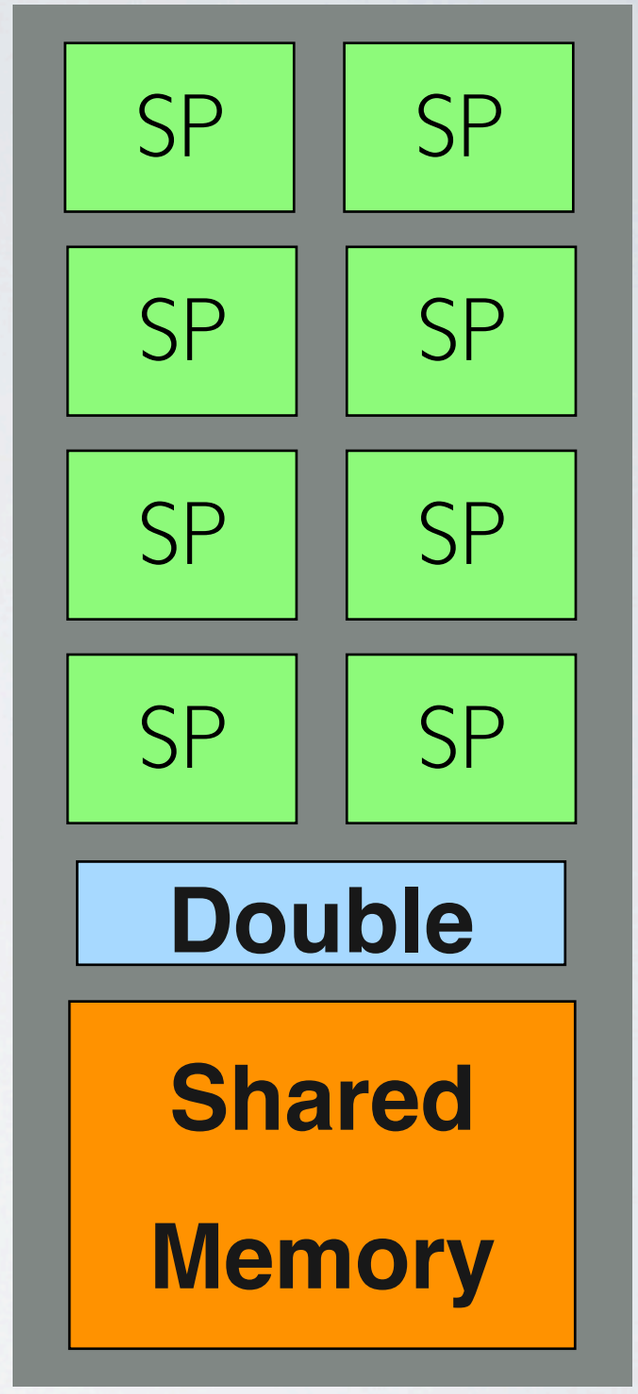
- CUDA C adds custom extensions to regular C to provide a much friendlier development environment compared to backdoor DirectX programming.
- Requirements
  - A CUDA-enabled graphics processor
    - Any NVIDIA GPU since 2006, starting with GeForce 8800 GTX
  - Free Downloads from NVIDIA for PC, Linux, or Mac
    - NVIDIA device driver
    - CUDA development toolkit
    - Standard C compiler
- CUDA C provides support for GPU programming and memory transfers with the nvcc compiler.
- A standard C compiler such as gcc does the rest.

# *THREADS*

- A Kernel is the sequence of instructions to be executed by each thread.
- From a programming point of view, a thread is just an execution of a kernel with a given index. Each thread has access to a one, two or three dimensional index which distinguishes it from other threads.
- The index can be used to access unique elements in an array, for example.
- Threads are parallel programs which can be thought of as executing the same code on different data.

# *BLOCKS AND MULTIPROCESSORS*

- The GPU hardware is organized as a collection of Multiprocessors.
- Each Multiprocessor handles one or more Blocks.
- A Multiprocessor is further divided into Stream Processors (also called Cores), with each Stream Processor handling one or more threads in a Block.
- Instruction execution flow is much more tightly coupled than with conventional CPU threads.



# *CUDA C SYNTAX*

- The additional syntax added to the C language is easy to learn, see here for some good examples and tutorial.
- the allocate, free and copy device memory:
  - cudaMalloc, cudaFree, cudaMemcpy
- define a device function (kernel) called "add" that operates on two integer arrays:
  - __global__ void add( int *a, int *b);

# *CUDA C SYNTAX CONTINUED*

- To execute 16 blocks, each of which runs the add function kernel, on elements of two integer arrays:
  - add<<< 16, 1 >>>(dev_a, dev_b)
- The second parameter inside the triple chevron specifies the number of threads to run per block.
  - add<<< 16, 32 >>>( ... ) runs 16*32 = 512 threads.
- Start and stop event recorders to profile your device code:
  - cudaEventRecord
  - cudaEventSynchronize
  - cudaEventElapsedTime

# *MEMORY*

- A GPU has different types of memory that can be used to optimize performance.
- Device memory is the memory on the same board as the GPU, usually a few GB. It is accessed with a very wide bus to support fast streaming.
- DMA can be used to transfer Device memory to the main Host memory.
- "Pinning" can be used to prevent disk caching of memory blocks by the host, but virtual memory performance will suffer.
- Constant Memory is a section of internal GPU memory which is read-only by the GPU. It can be used to store data that will not change during thread execution.

# *MEMORY CONTINUED*

- Shared Memory is fast memory that shared between threads within the same block, limited to around 64K or so.
- Texture Memory is fast read only memory that is optimized to take advantage of two dimensional access patterns. Designed for the graphics pipeline, it is still claimed to be useful for many practical general purpose algorithms.

# *ATOMICS*

- Atomics are provided to coordinate memory accesses among threads and prevent race conditions.
- For the fast, shared memory in a CUDA GPU, this means threads within the same block.
- Atomics can also be used to protect global memory accesses, which will be relatively slow.
- Atomic types include Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor.
- Example:
  - unsigned int atomicInc(unsigned int* address, unsigned int val);

# *SYNCHRONIZATION*

- Threads within a block can be synchronized, so all will pause execution until all are finished.
- Synchronization is only possible between threads in the same Block.
- Synchronization is accomplished with the single line: __syncthreads();
- If __syncthreads() is executed in a kernel within a conditional construct, like if or while, the GPU could freeze.
- This is because all threads are running from the same program control unit, and progress will halt until all threads execute the __syncthreads() line.
- But because some threads may not pass the condition, they will never get there, and this results in a nasty halt condition.

# *STREAMS*

- Streams make it possible to define sequences of operations which run asynchronously on the GPU.
  - Allows the host to operate concurrently with the GPU.
  - Fairly intuitive for programmers.
- Multiple streams can be run concurrently on the GPU to allow memory copy to occur in parallel with computation for multiple buffers.
- Streaming requires some deeper understanding about the GPU hardware to get it right.
  - Some GPUs don't allow parallelizing memory operations and kernel programs.
  - Some allow one block read only, one block write only and one kernel to be  simultaneous.
  - High end GPU's allow multiple Kernels, too.

# *OTHER CUDA CAPABILITIES*

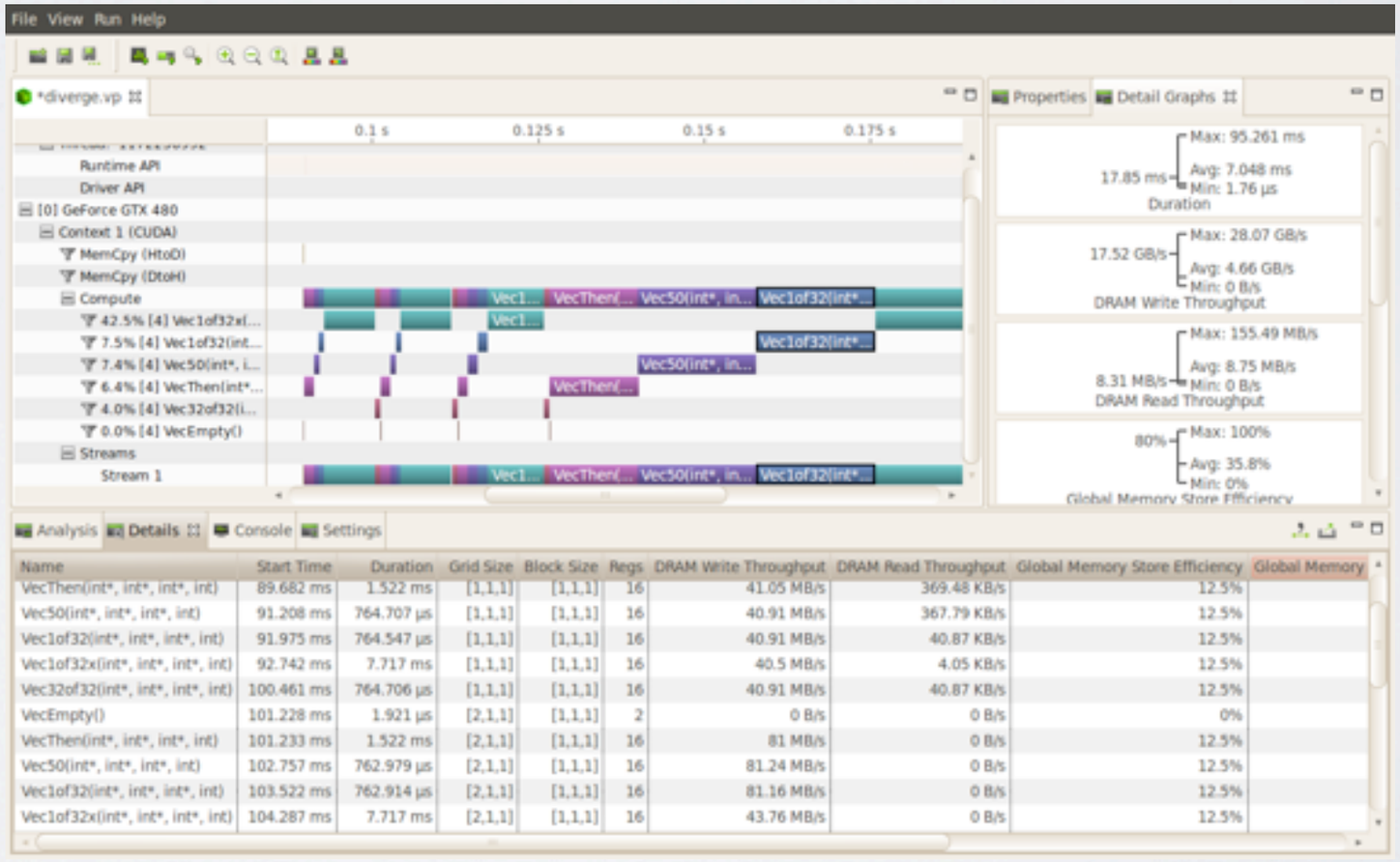
- CUDA allows multiple GPUs to be used within the same system.
- It also allows the GPU to be used as a graphics device in parallel with using the device for general computation.
- This capability is used by some game programmers to accelerate the physics engines while rendering the graphics of the game.

# *DEBUGGER*

- Parallel NSight is a visual debugger for Windows developers.
- CUDA gdb is the command line debugger for mac and linux.
- Both offer remote debugging and standard functions like breakpoints, single step, etc.
- Adds information about devices, warps, lanes, kernels, blocks SM's and threads.
- Single stepping GPU code is at the "Warp" level, and advances all threads together..
- Keeping track of kernels, grids, blocks,stream multiprocessors, threads, etc. with a command line debugger doesn't sound easy but I haven't tried it yet.

# *VISUAL PROFILER*

- Release for Mac, Linux and Windows in Jan 2012.
- Automated Performance Analysis
- Unified CPU/GPU Timeline
- CUDA and OpenCL API trace




| Name | Start Time | Duration | Grid Size | Block Size | Regs | DRAM Write Throughput | DRAM Read Throughput | Global Memory Store Efficiency | Global Memory |
|---|---|---|---|---|---|---|---|---|---|
| VecThen(int*, int*, int*, int) | 89.682 ms | 1.522 ms | [1,1,1] | [1,1,1] | 16 | 41.05 MB/s | 369.48 KB/s | 12.5% | |
| Vec50(int*, int*, int*, int) | 91.208 ms | 764.707 µs | [1,1,1] | [1,1,1] | 16 | 40.91 MB/s | 367.79 KB/s | 12.5% | |
| Vec1of32(int*, int*, int*, int) | 91.975 ms | 764.547 µs | [1,1,1] | [1,1,1] | 16 | 40.91 MB/s | 40.87 KB/s | 12.5% | |
| Vec1of32x(int*, int*, int*, int) | 92.742 ms | 7.717 ms | [1,1,1] | [1,1,1] | 16 | 40.5 MB/s | 4.05 KB/s | 12.5% | |
| Vec32of32(int*, int*, int*, int) | 100.461 ms | 764.706 µs | [1,1,1] | [1,1,1] | 16 | 40.91 MB/s | 40.87 KB/s | 12.5% | |
| VecEmpty() | 101.228 ms | 1.921 µs | [2,1,1] | [1,1,1] | 2 | 0 B/s | 0 B/s | 0% | |
| VecThen(int*, int*, int*, int) | 101.233 ms | 1.522 ms | [2,1,1] | [1,1,1] | 16 | 81 MB/s | 0 B/s | 12.5% | |
| Vec50(int*, int*, int*, int) | 102.757 ms | 762.979 µs | [2,1,1] | [1,1,1] | 16 | 81.24 MB/s | 0 B/s | 12.5% | |
| Vec1of32(int*, int*, int*, int) | 103.522 ms | 762.914 µs | [2,1,1] | [1,1,1] | 16 | 81.16 MB/s | 0 B/s | 12.5% | |
| Vec1of32x(int*, int*, int*, int) | 104.287 ms | 7.717 ms | [2,1,1] | [1,1,1] | 16 | 43.76 MB/s | 0 B/s | 12.5% | |

# *PROGRAMMING LIBRARIES*

- CUDA Parallel FFT Library (CUFFT)
  - 7-18x performance claimed
- CUDA Basic Linear Algebra Subprograms (CUBLAS)
- CUDA LAPACK aka Magma development co-sponsored by CU Denver.
- math.h for CUDA
- CUDA Random Library
- NVidia Performance Primitives (NPP)
  - 350 image processing functions
  - 100 signal processing functions

# *MORE INFO*

- CUDA Training | NVIDIA Developer Zone
- CUDA by Example, Jason Sanders and Edward Kandrot, 201
- stanford-cs193g-sp2010 - Programming Massively Parallel Processors with CUDA - Google Project Hosting
- CUDA, Supercomputing for the Masses | Dr Dobb's
- Triers CUDA ray tracing tutorial « Computer Graphics Lab

# *REFERENCES*


- GPU Programming in MATLAB - MathWorks Newsletter
- "Understanding performance bottlenecks in numerical kernels on GPUs" by Vasily Volkov, May 21, 2010,.
- http://www.pcmag.com/article2/0,2817,2374890,00.asp
- http://drdobbs.com/architecture-and-design/207200659
- Hot-Rodding Windows and Linux App Performance with CUDA-Based Plugins | Dr Dobb's
- http://www.pgroup.com/lit/articles/insider/v2n1a5.htm
- http://www.mothdesigns.co.uk/content/nvidia-fermi-fail/

# *REFERENCES*


- GPU AI for Board Games | NVIDIA Developer Zone
- Imaging and Computer Vision
- NVIDIA Tesla Success Stories
- http://www.nvidia.com/object/evolved_machines_neural_circuit.html
- http://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html
- Psychlone Personal Supercomputer
- A Neural Network on GPU - CodeProject®
- Kyle Niemeyer » GPU programming with CUDA on MacBook Pro