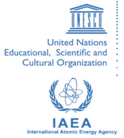
The Abdus Salam
International Centre for Theoretical Physics



2068-18

Advanced School in High Performance and GRID Computing - Concepts and Applications

30 November - 11 December, 2009

Introduction to Graphical Processing Units as Tools for Scientific Computing

Ben Levine

Temple University
Philadelphia
USA

Strada Costiera 11, 34151 Trieste, Italy - Tel.+39 040 2240 111; Fax +39 040 224 163 - sci_info@ictp.it

Introduction to Graphical Processing Units as Tools for Scientific Computing

Benjamin G. Levine

Institute for Computational Molecular Science
and Department of Chemistry
Temple University
Philadelphia, PA, USA

Hopefully this talk will (partially) answer…

- What is a GPU?
- Why would I use a GPU for scientific computing?
- When should I consider using GPUs to solve my problem?
- What are the main hardware features available to the GPU programmer?
- How do I use the GPU to solve my problem?
- How do I use the GPU to solve my problem <u>really fast</u>?

What is a graphical processing unit (GPU)?

- A processor you add to your computer to accelerate graphical applications, e.g. computer games
- Most desktops and laptop computers already contain a GPU of some kind

http://www.nvnews.net/previews/geforce_8800_gtx/images/geforce_8800_gts.jpg

Why do we want to use GPUs for scientific computing?

- They are inexpensive
- They are readily available, and possibly already present in your desktop workstation
- They are fast for floating point math
- The operations involved in game software are similar to those in scientific applications

Processor	Cost	Floating Point Performance	Memory Bandwidth
Quad-Core Intel Xeon E5506 @2.13GHz	$1500	68 Gflop/s	19 GB/s
NVidia GTX 275	+$300	304 Gflop/s	127 GB/s

GPUs outperform CPUs on a per dollar basis

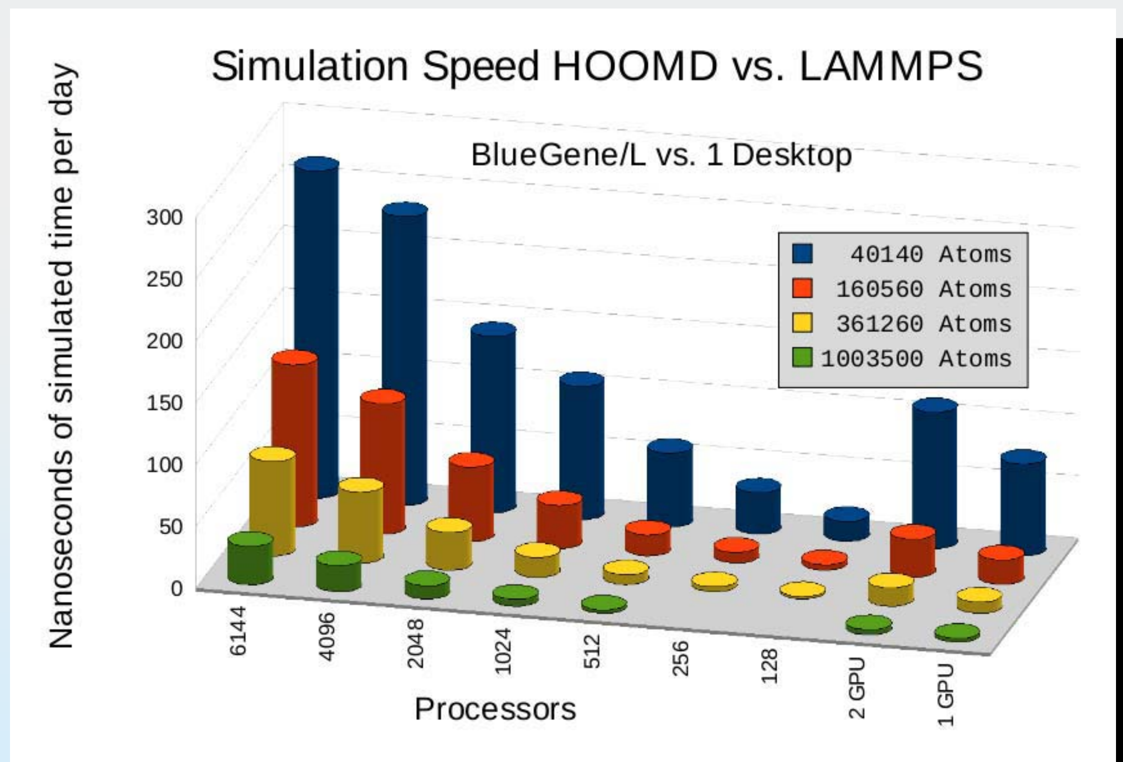
How is a game similar to a physics simulation?

- Games <u>are</u> physic simulations
- Graphical manipulations are linear algebra

Physics and games both require fast floating point math

How fast?

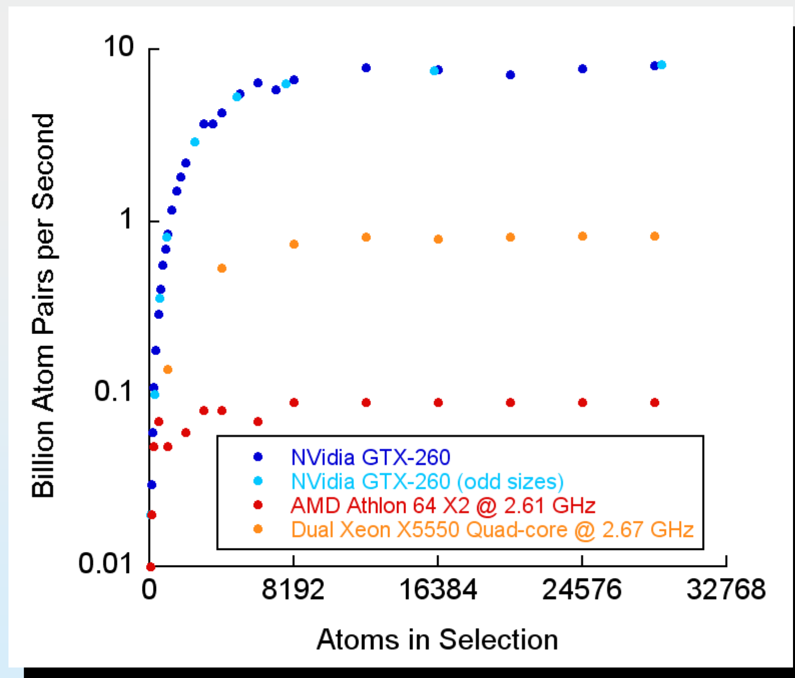
- HOOMD J. A. Anderson, et al. J Comp. Phys. 227 (2008) 5342

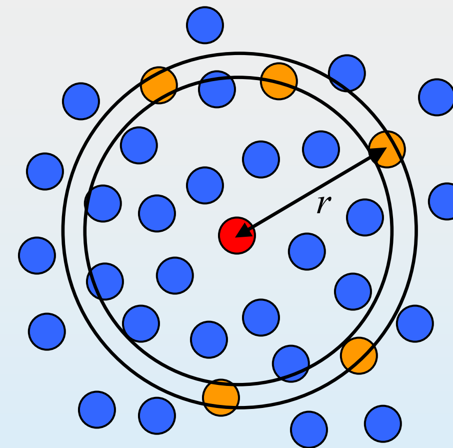


Thanks to Axel Kohlmeyer and David LeBard

How fast?

- Radial Distribution Functions

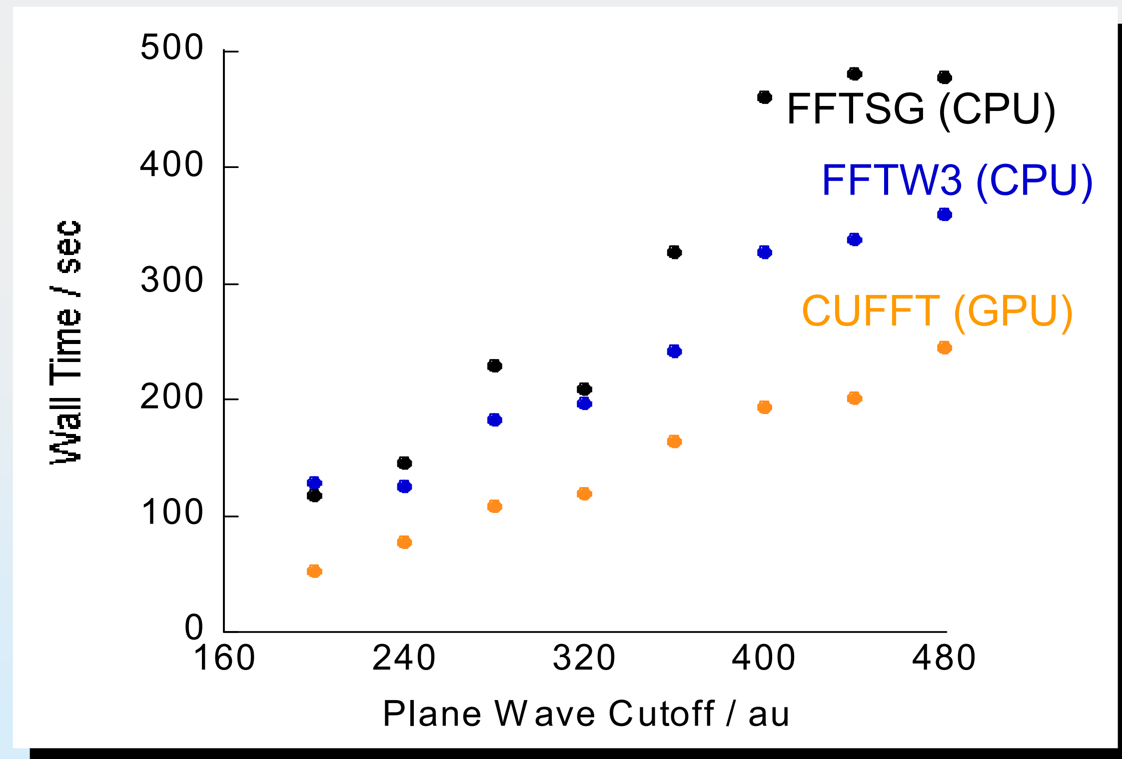




How fast?

- CP2K

Density functional calculation - A single water molecule in a 15 Å periodic box, Pade functional

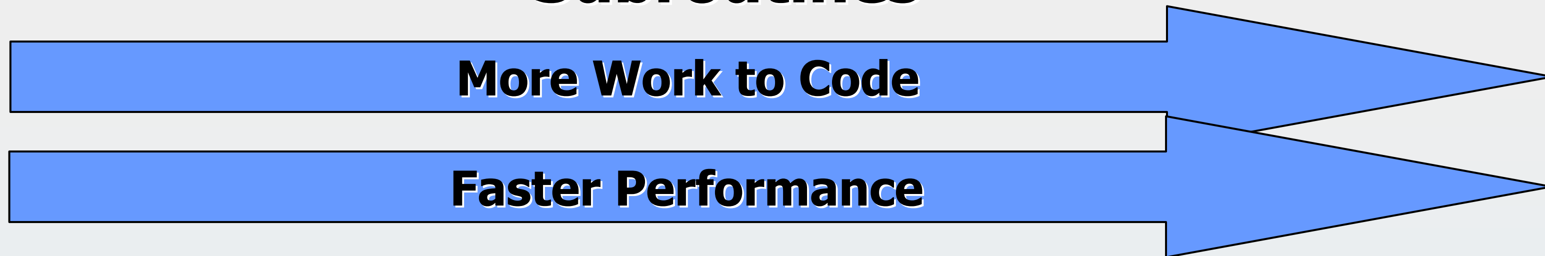


GPU Acceleration Strategies

Link to Libraries

Hand-code Performance-Critical Subroutines

Rewrite Whole Code From Scratch



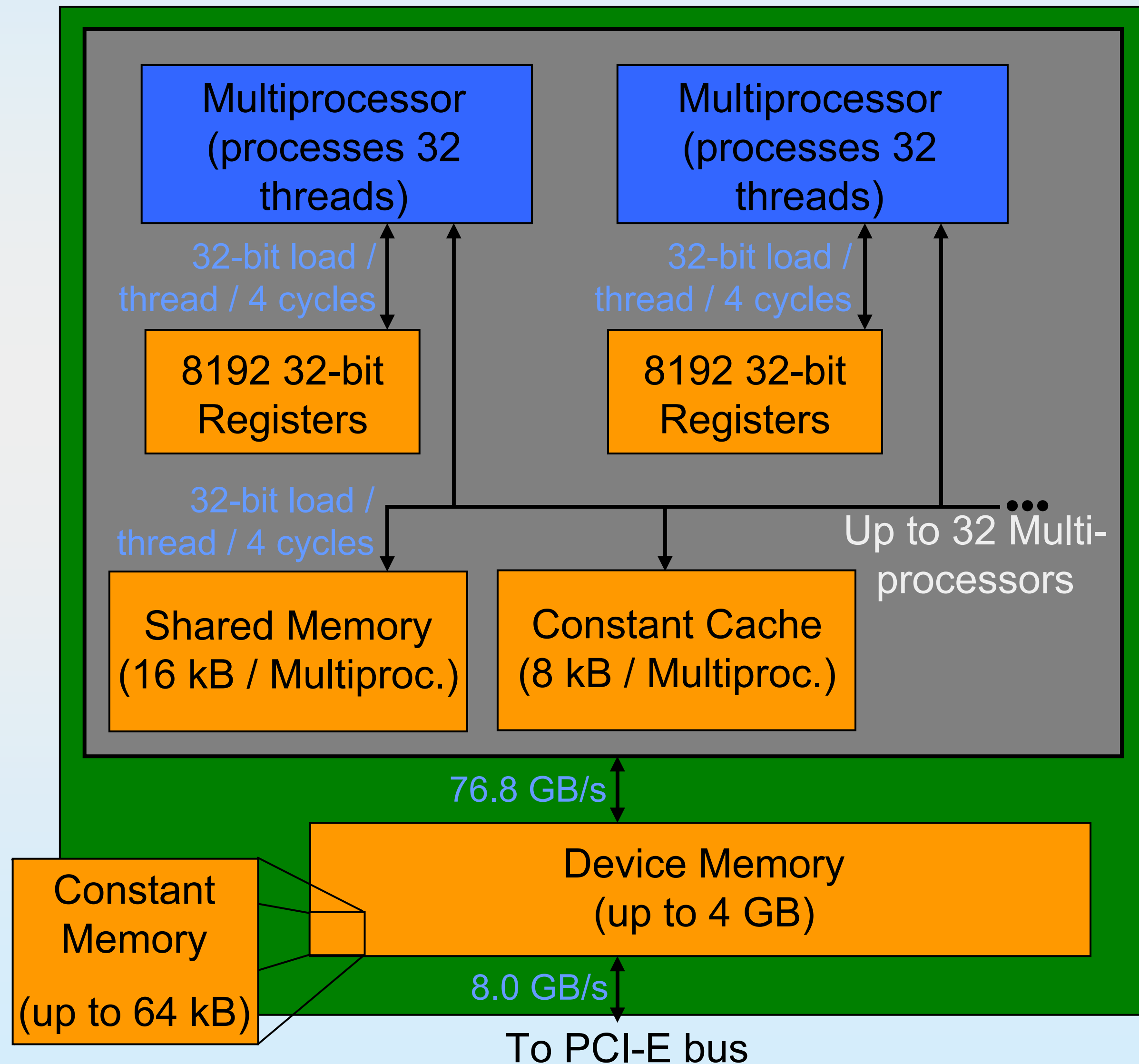
CP2K
(2-3x speedup)

HOOMD
(30x speedup)

Radial Distribution Functions
(10-100x speedup)

NVidia GPU Architecture

- Parallelism
- Memory Hierarchy

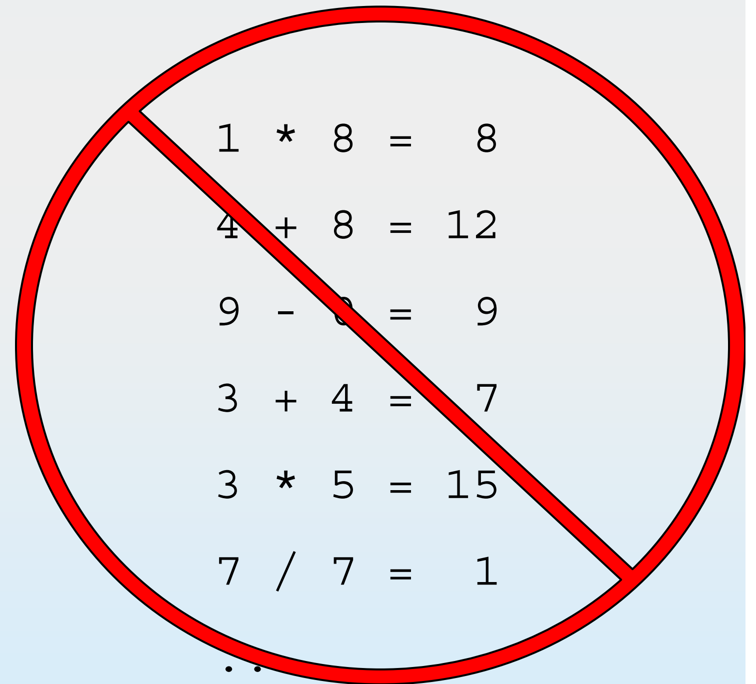


NVidia GPU Architecture

- Applies the same instruction to 32 pieces of data simultaneously (SIMD)

Multiprocessor (processes 32 threads)

```
1 + 8 =  9
4 + 8 = 12
9 + 0 =  9
3 + 4 =  7
3 + 5 =  8
7 + 7 = 14
...
```



NVidia GPU Architecture

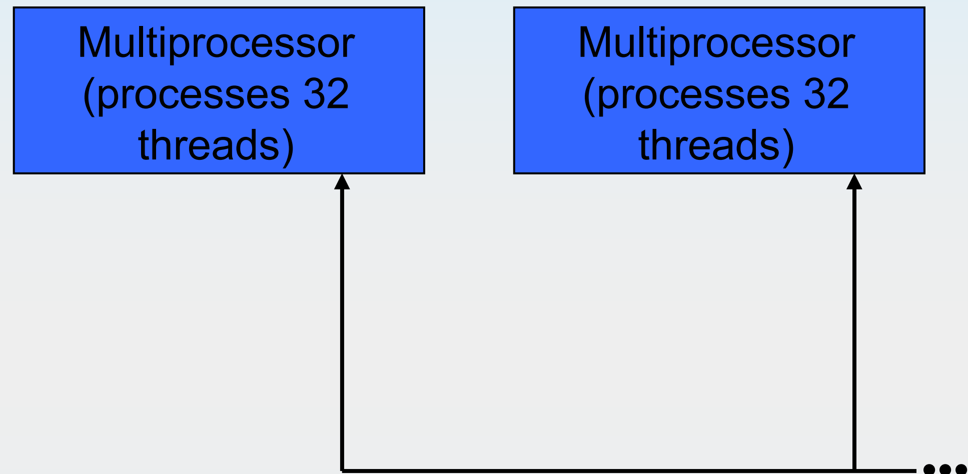
- Double precision floating point math is 7x slower than single (next generation "Fermi" will be only 2x slower)
- Fused multiply-add

Multiprocessor (processes 32 threads)

```
1 * 2 + 8 = 10
4 * 3 + 8 = 20
9 * 1 + 0 =  9
3 * 8 + 4 = 28
3 * 3 + 5 = 14
7 * 4 + 7 = 35
...
```

NVidia GPU Architecture

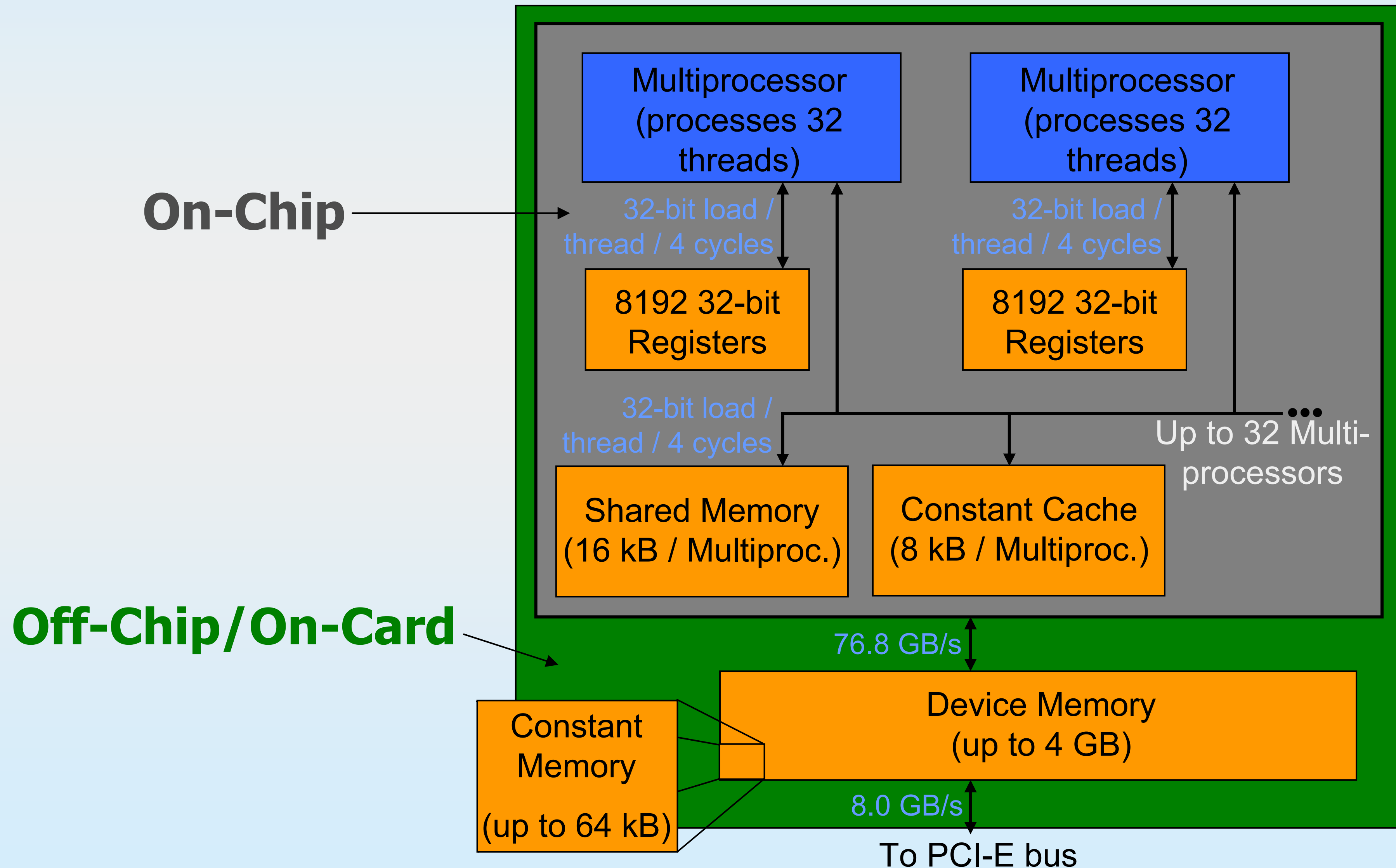
- A GPU chip contains between 1 and 32 multiprocessors



32 multiprocessors * 32 threads / multiprocessor * 2 FLOPs per cycle = 2048 FLOPS at one time!

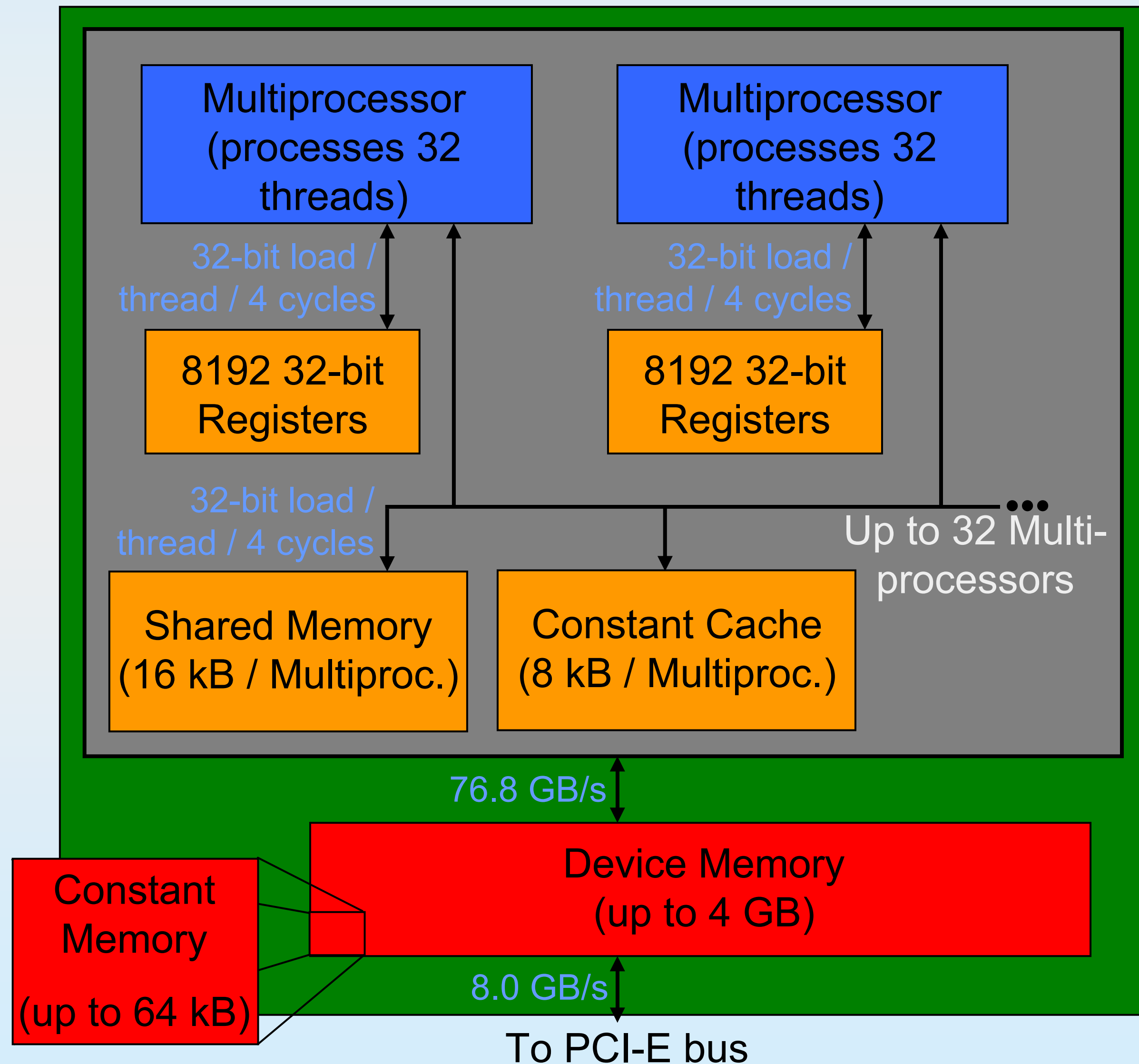
The key to successful GPU programming is keeping all multiprocessors busy as much as is possible

NVidia GPU Architecture



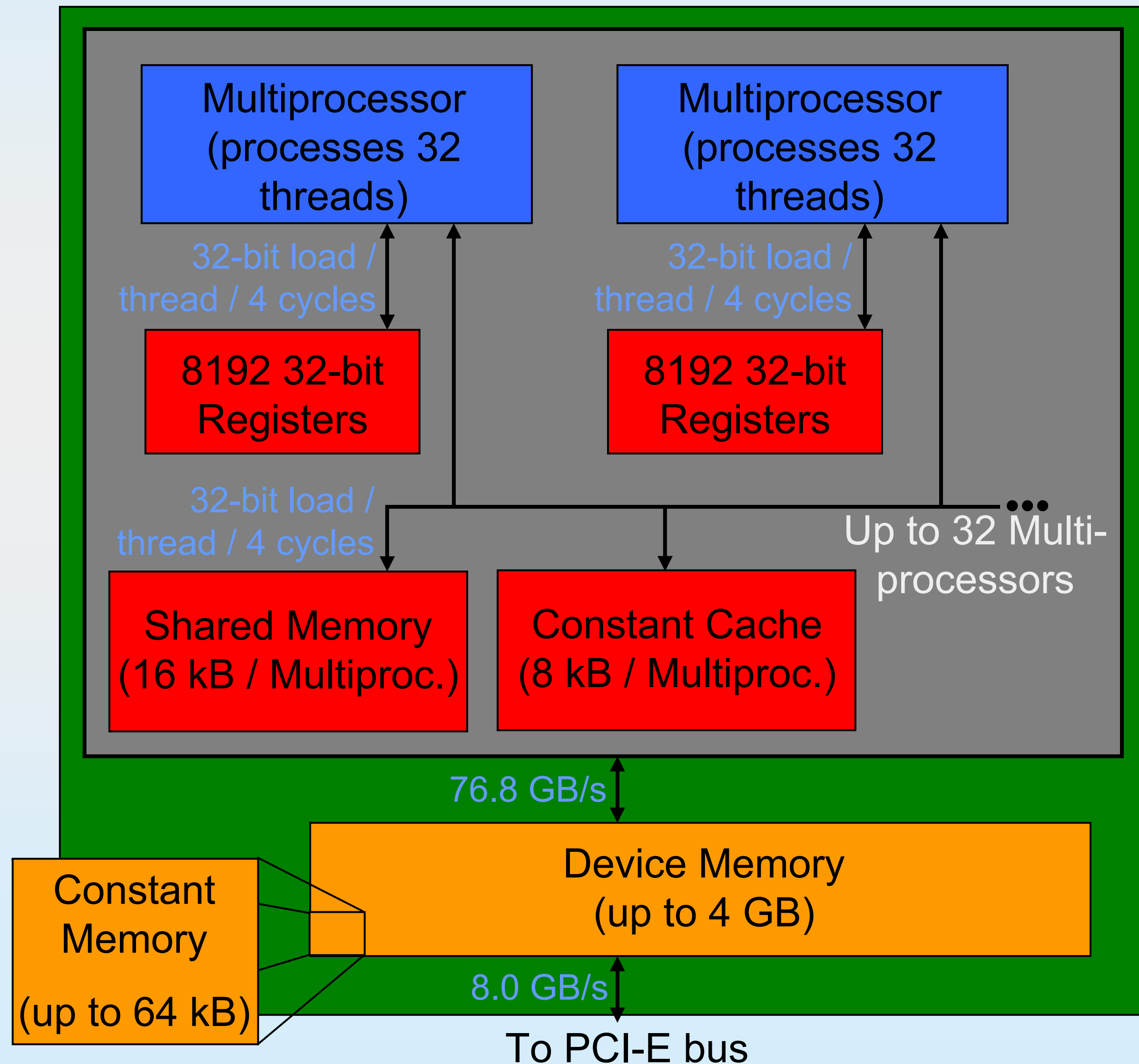
NVidia GPU Architecture

- Slow off-chip memory



NVidia GPU Architecture

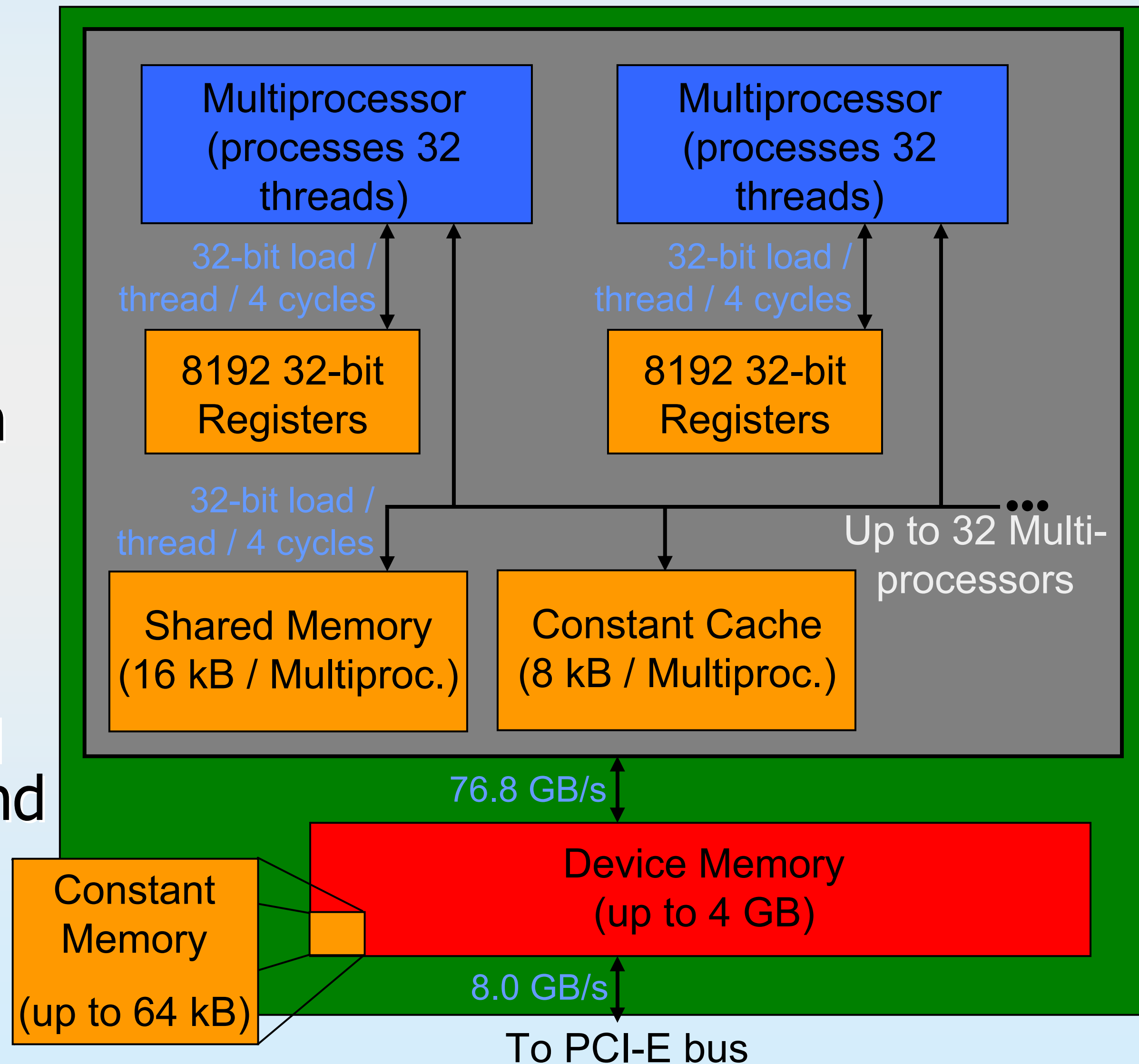
- Slow off-chip memory
- Fast on-chip memory



NVidia GPU Architecture

Device (Global) Memory

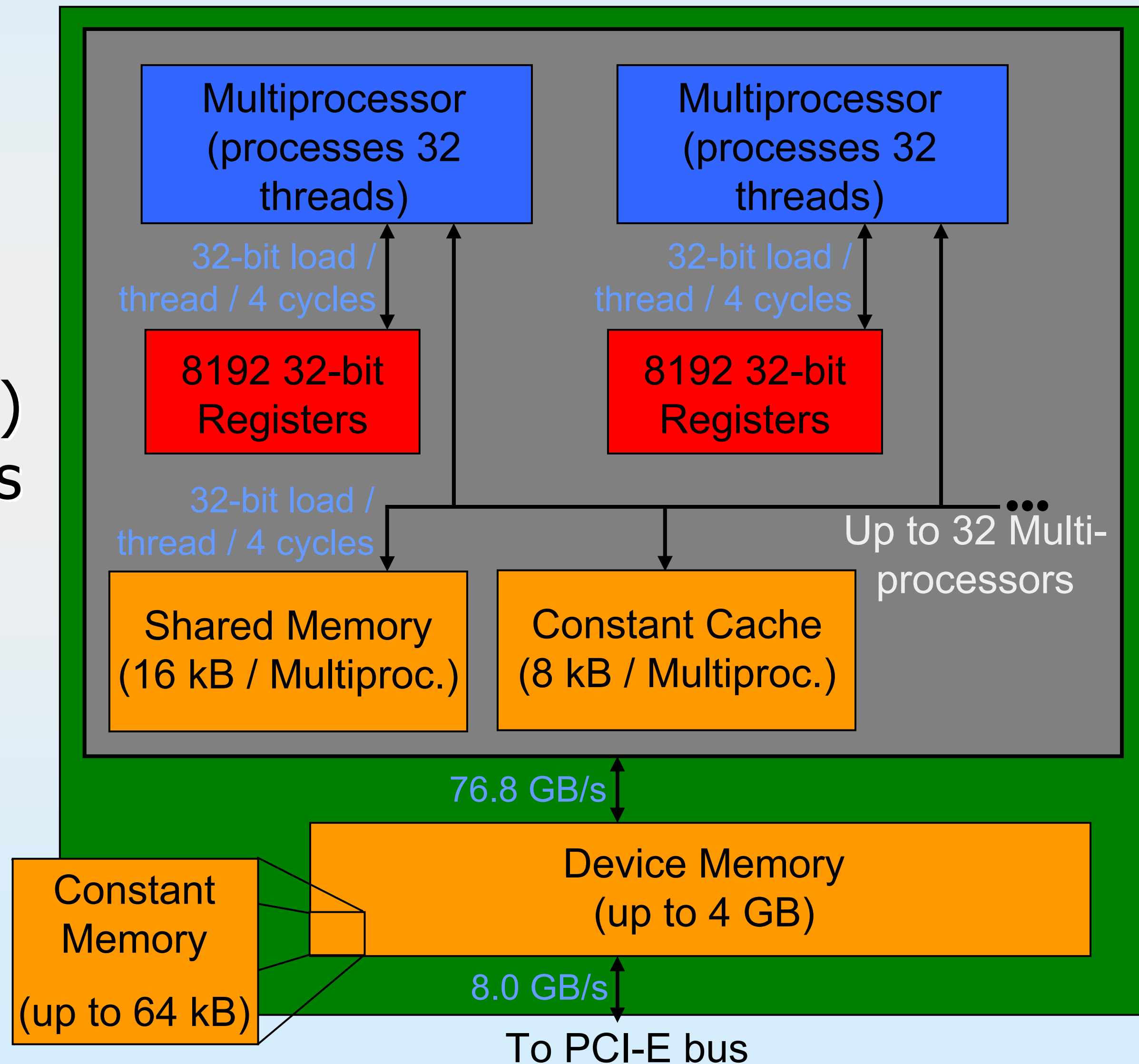
- Slow
- Large
- Analogous to main memory in a desktop computer
- Typically not cached
- Accessible from all threads on GPU and from the CPU



NVidia GPU Architecture

Registers

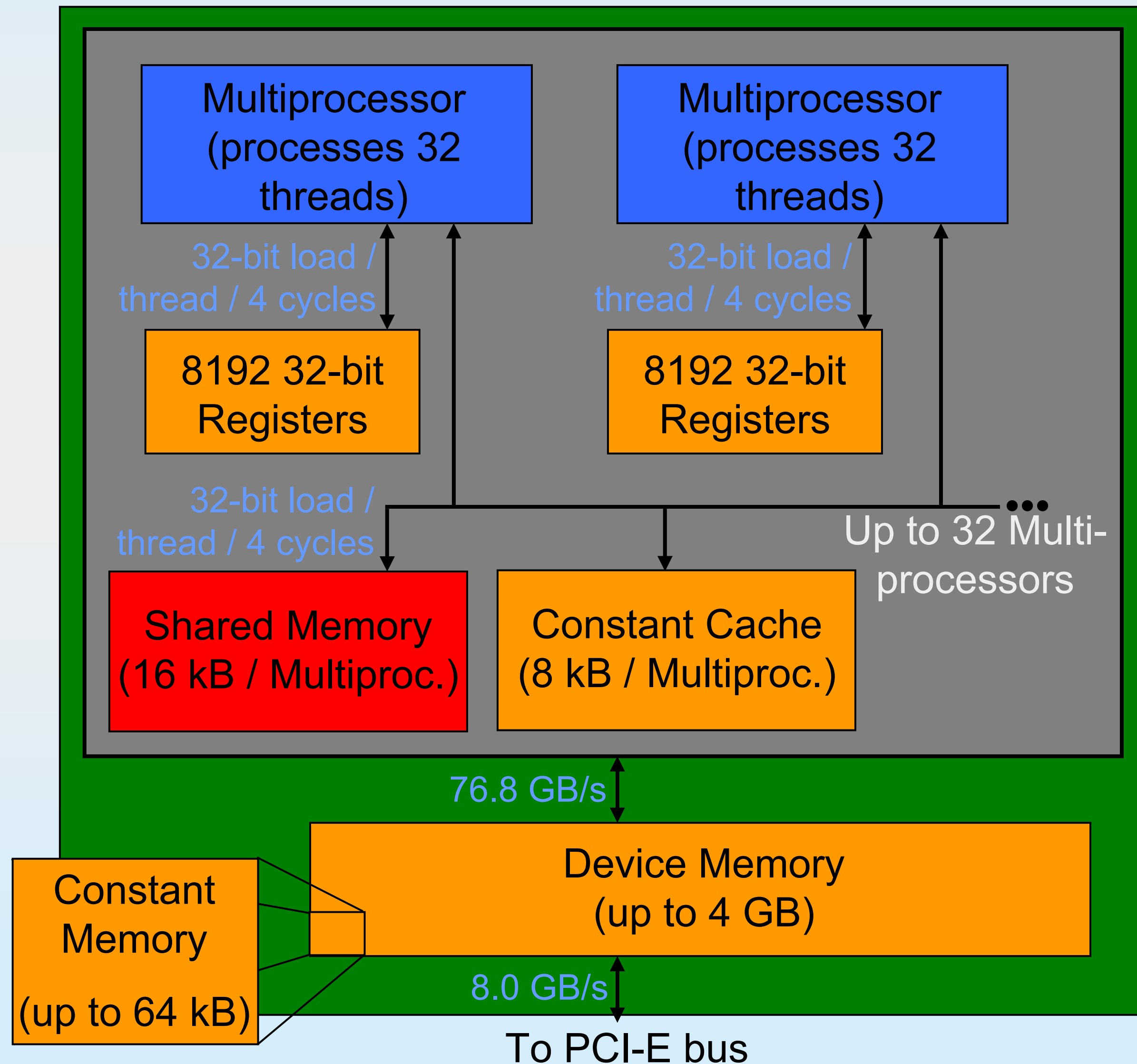
- Loads very fast
- Limited supply (~25/thread for good performance)
- Each thread has its own
- Analogous to registers in a computer
- Can be accessed only from the GPU



NVidia GPU Architecture

Shared memory

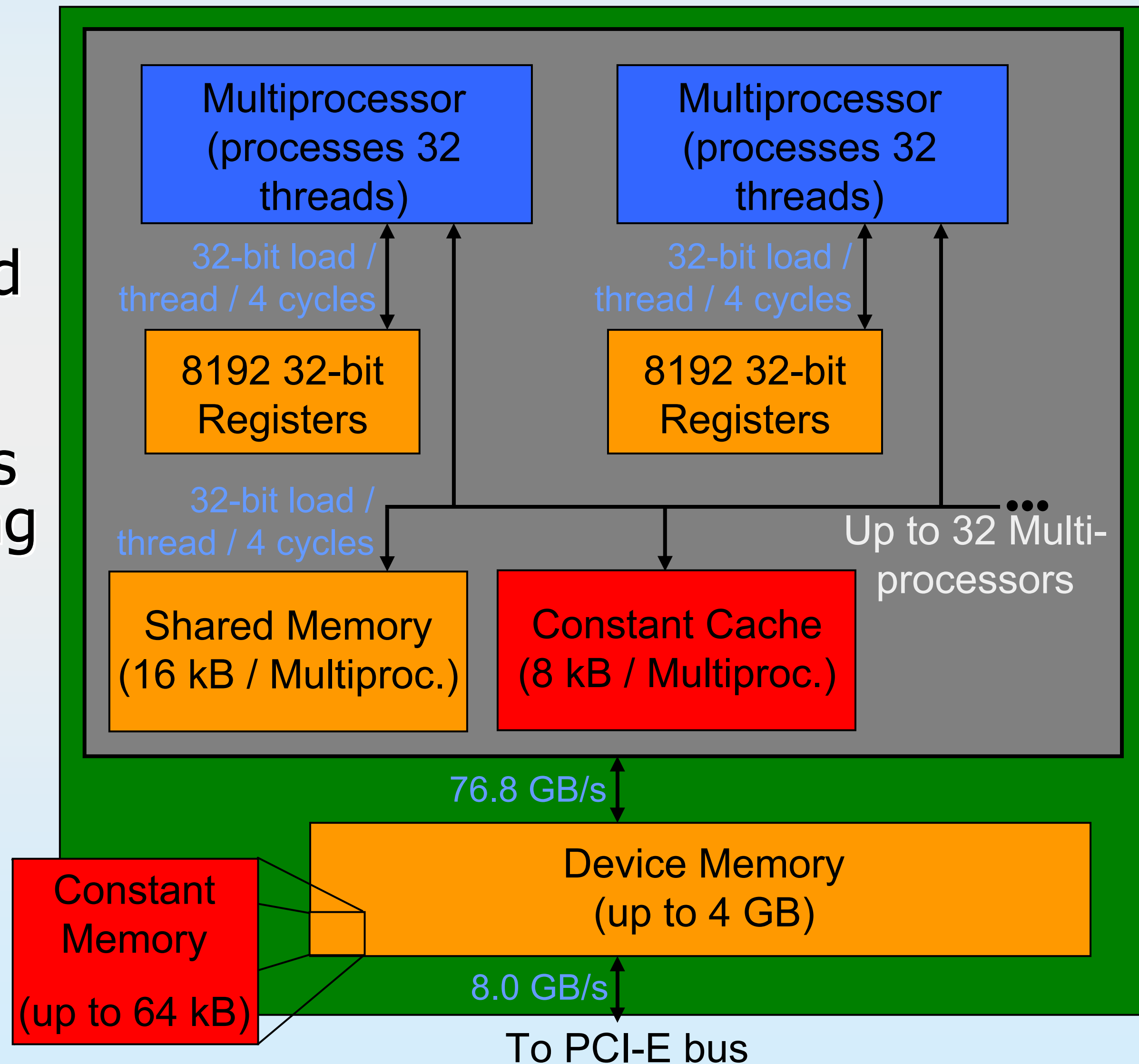
- *Can* be as fast as registers
- Limited supply
- Like a cache, but user controlled
- Shared by several threads
- Can only be accessed by the GPU



NVidia GPU Architecture

Constant Memory

- Device memory which is associated with an on-chip cache
- As fast as registers if you are accessing cached data
- Limited supply
- Read-only from GPU
- Read-write from CPU



Memory Hierarchy Summary

Layers of memory differ in

- size
- speed
- whether they shared between among several threads
- whether they can be written to from the GPU or only read
- whether they are associated with a cache

The programmer has almost complete control over the location of data in this hierarchy!

What we've learned so far…

- GPUs are fast
- GPUs are relatively inexpensive
- The more time you put into programming, the faster your GPU code can be
- GPUs are highly parallel processors
- GPU memory is broken down into a large amount of slow memory, and several small chunks of fast memory

But what do we do if we want to use it?

First, stop and THINK

- How much time do I have to code? – A lot of coding is needed to gain a large speedup in most cases.
- Which parts of my code are performance critical? – Perhaps optimizing a single routine could give a substantial speedup.
- Will my problem map well to the GPU?

Will my problem map well to the GPU?

- Those which involve:
  - performing the same operations on a lot of data
  - lots of floating point math
  - regular memory access pattern
  - problem which can be solved (mostly) in single precision
- Examples
  - algorithms based on linear algebra
  - parallel random number generation (MC with multiple walkers)

Compute Unified Device Architecture (CUDA)

- Created by NVidia to allow their GPUs to be used for general high-performance computing applications
- An extension to the C programming language
- Programming Guide: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf
- To learn more about applications and find online tutorials: http://www.nvidia.com/object/cuda_education.html

Installing CUDA

- Download from: http://www.nvidia.com/object/cuda_home.html#
- Download and install three things
  - CUDA driver – tells your computer how to access the GPU hardware
  - CUDA toolkit – contains the NVCC compiler and libraries needed to compile GPU accelerated code
  - CUDA standard developers kit (SDK) – contains example codes which are useful in learning CUDA
- Easiest to install with one of the supported Linux distributions (Redhat Enter., Fedora, SUSE Enter., OpenSUSE, Ubuntu)

Compiling CUDA Code

- Include the cuda_runtime header file at the beginning of your source file

```
#include <cuda_runtime.h>
```

- Use nvcc just like any other compiler

```
nvcc -o xyz.e xyz.cu
```

- Preprocessing is the same as for C
- IMPORTANT – simply adding the above `#include` statement and compiling with `nvcc` does not result in any portion of your code being run on the GPU!

Emulation Mode

- Emulation mode executables run GPU code on the CPU only
- Emulation mode is particular useful for debugging because input/output (e.g. `printf`) is not available on the GPU
- To compile in emulations mode:

```
nvcc -o xyz.e -deviceemu -D__DEVICEEMU xyz.cu
```

- Then, to print

```
#ifdef __DEVICEEMU
printf("debugging info");
#endif
```

Math Libraries

- CUBLAS – Linear Algebra
  - Single and Double Precision BLAS Routines
  - Helper functions
    - cublasInit and cublasShutdown must be run before and after any calls
    - Routines to allocate and free device memory
    - Routines to move data between main memory and device
  - http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf
- CUFFT – Fast Fourier Transforms
  - Interface similar to FFTW
  - http://developer.download.nvidia.com/compute/cuda/1_1/CUFFT_Library_1.1.pdf

Linking to the Math Libraries

- To link to the CUBLAS or CUFFT libraries include the appropriate header file in your source code

```
#include <cublas.h>
#include <cufft.h>
```

- Pass the option to link to the appropriate library to nvcc

```
nvcc -o xyz.e -lcublas xyz.cu
nvcc -o xyz.e -lcufft xyz.cu
```

Terminology

- Host = the non-GPU part of your machine
  - Host memory = main memory
  - Host processor = CPU
- Device = the graphics card
  - Device memory = the off-chip memory on the device (global memory)

What Does CUDA Code Look Like?

- The following code squares each element of a vector `A_host` on the CPU

```
void square_vec(float* A_host,
                int n) {
  for (int i=0; i<n; i++) {
    A_host[i] = A_host[i] *
        A_host[i];
  }
}
```

What Does CUDA Code Look Like?

- The main routine looks almost like standard C
- It is called from and executed on the CPU

```
void square_vec(float* A_host,
                int n) {
  float* A_dev;
  cudaMalloc((void**)&A_dev,
      n * sizeof(float));
  cudaMemcpy(A_dev, A_host,
      n * sizeof(float),
      cudaMemcpyHostToDevice);
  gpu_kernel<<<1,n>>>(A_dev);
  cudaMemcpy(A_host, A_dev,
      n * sizeof(float),
      cudaMemcpyDeviceToHost);
  cudaFree(A_dev);
}
```

What Does CUDA Code Look Like?

`A_host` is a vector of length `n` stored in the host (main) memory

This code with calculate

`A_host[i] = A_host[i]^2`

for all elements of `A_host` in parallel on the GPU

```
void square_vec(float* A_host,
                int n) {
  float* A_dev;
  cudaMalloc((void**)&A_dev,
      n * sizeof(float));
  cudaMemcpy(A_dev, A_host,
      n * sizeof(float),
      cudaMemcpyHostToDevice);
  gpu_kernel<<<1,n>>>(A_dev);
  cudaMemcpy(A_host, A_dev,
      n * sizeof(float),
      cudaMemcpyDeviceToHost);
  cudaFree(A_dev);
}
```

What Does CUDA Code Look Like?

Enough global memory to store the array on the device is dynamically allocated using the `cudaMalloc` library routine

```
void square_vec(float* A_host,
                int n) {
  float* A_dev;
  cudaMalloc((void**)&A_dev,
      n * sizeof(float));
  cudaMemcpy(A_dev, A_host,
      n * sizeof(float),
      cudaMemcpyHostToDevice);
  gpu_kernel<<<1,n>>>(A_dev);
  cudaMemcpy(A_host, A_dev,
      n * sizeof(float),
      cudaMemcpyDeviceToHost);
  cudaFree(A_dev);
}
```

What Does CUDA Code Look Like?

The contents of `A_host` (in main memory) are copied to `A_dev` (in device memory) using the `cudaMemcpy` library routine

```
void square_vec(float* A_host,
                int n) {
  float* A_dev;
  cudaMalloc((void**)&A_dev,
      n * sizeof(float));
  cudaMemcpy(A_dev, A_host,
      n * sizeof(float),
      cudaMemcpyHostToDevice);
  gpu_kernel<<<1,n>>>(A_dev);
  cudaMemcpy(A_host, A_dev,
      n * sizeof(float),
      cudaMemcpyDeviceToHost);
  cudaFree(A_dev);
}
```

What Does CUDA Code Look Like?

`gpu_kernel` is a user written routine which runs on the GPU.

The bracketed numbers `<<<1,n>>>` denote that `1` block of `n` threads will be spawned.

Blocks of threads share shared memory and can be synchronized with barriers; threads in different blocks cannot share memory or be syncronized.

```
void square_vec(float* A_host,
                int n) {
  float* A_dev;
  cudaMalloc((void**)&A_dev,
      n * sizeof(float));
  cudaMemcpy(A_dev, A_host,
      n * sizeof(float),
      cudaMemcpyHostToDevice);
  gpu_kernel<<<1,n>>>(A_dev);
  cudaMemcpy(A_host, A_dev,
      n * sizeof(float),
      cudaMemcpyDeviceToHost);
  cudaFree(A_dev);
}
```

What Does CUDA Code Look Like?

After `gpu_kernel` completes, the contents of `A_dev` (in device memory) are copied to `A_host` (in main memory) using the `cudaMemcpy` library routine

```
void square_vec(float* A_host,
                int n) {
  float* A_dev;
  cudaMalloc((void**)&A_dev,
      n * sizeof(float));
  cudaMemcpy(A_dev, A_host,
      n * sizeof(float),
      cudaMemcpyHostToDevice);
  gpu_kernel<<<1,n>>>(A_dev);
  cudaMemcpy(A_host, A_dev,
      n * sizeof(float),
      cudaMemcpyDeviceToHost);
  cudaFree(A_dev);
}
```

What Does CUDA Code Look Like?

```
void square_vec(float* A_host,
                int n) {
  float* A_dev;
  cudaMalloc((void**)&A_dev,
      n * sizeof(float));
  cudaMemcpy(A_dev, A_host,
      n * sizeof(float),
      cudaMemcpyHostToDevice);
  gpu_kernel<<<1,n>>>(A_dev);
  cudaMemcpy(A_host, A_dev,
      n * sizeof(float),
      cudaMemcpyDeviceToHost);
  cudaFree(A_dev);
}
```

The dynamically allocated device memory pointed to by `A_dev` is freed

What Does CUDA Code Look Like?

```
void square_vec(float* A_host,
                int n) {
  float* A_dev;
  cudaMalloc((void**)&A_dev,
      n * sizeof(float));
  cudaMemcpy(A_dev, A_host,
      n * sizeof(float),
      cudaMemcpyHostToDevice);
  gpu_kernel<<<1,n>>>(A_dev);
  cudaMemcpy(A_host, A_dev,
      n * sizeof(float),
      cudaMemcpyDeviceToHost);
  cudaFree(A_dev);
}
```

What Does CUDA Code Look Like?

Define the function with the `__global__` function type qualifier to tell the compiler that this function will be called from the host, but run on the device

```
__global__ void gpu_kernel(
                  float* A_dev) {
  float tmp;
  tmp = A_dev[threadIdx.x];
  tmp = tmp * tmp;
  A_dev[threadIdx.x] = tmp;
}
```

What Does CUDA Code Look Like?

Declaring `tmp` without a variable type qualifier indicates that we want the compiler to store `tmp` in a register if possible

If the compiler determines that it does not want to use a register for this variable it will be placed in global memory

```
__global__ void gpu_kernel(
               float* A_dev) {
  float tmp;
  tmp = A_dev[threadIdx.x];
  tmp = tmp * tmp;
  A_dev[threadIdx.x] = tmp;
}
```

What Does CUDA Code Look Like?

We load an element of the `A_dev` into the register (`tmp`)

`threadIdx.x` is a unique identifier for each thread of the block, starting at 0 and counting up to `blockDim.x-1`, thus we load a unique element of `A_dev` for each thread

```
__global__ void gpu_kernel(
                    float* A_dev) {
  float tmp;
  tmp = A_dev[threadIdx.x];
  tmp = tmp * tmp;
  A_dev[threadIdx.x] = tmp;
}
```

What Does CUDA Code Look Like?

Just multiplication, nothing special

```
__global__ void gpu_kernel(
                    float* A_dev) {
  float tmp;
  tmp = A_dev[threadIdx.x];
  tmp = tmp * tmp;
  A_dev[threadIdx.x] = tmp;
}
```

What Does CUDA Code Look Like?

Store the result of the multiplication from the register (`tmp`) back into global memory (`A_dev`)

```
__global__ void gpu_kernel(
                    float* A_dev) {
  float tmp;
  tmp = A_dev[threadIdx.x];
  tmp = tmp * tmp;
  A_dev[threadIdx.x] = tmp;
}
```

What Does CUDA Code Look Like?

```
__global__ void gpu_kernel(
                float* A_dev) {
  float tmp;
  tmp = A_dev[threadIdx.x];
  tmp = square(tmp);
  A_dev[threadIdx.x] = tmp;
}
```

If we want we can create `__device__` functions which are callable from the GPU and run on the GPU

```
__device__ float square(
                float x) {
  float y
  y = x * x;
  return y;
}
```

Summary of CUDA extensions to standard C

- Memory Management
  - `cudaMalloc` – Allocates global memory on the device
  - `cudaMemcpy` – Copies memory from host to device and from device to host
  - `cudaFree` – Frees dynamically allocated global memory

Summary of CUDA extensions to standard C

- Function type qualifiers
  - `__host__` or no type qualifier – runs on the host and is callable only from the host
  - `__global__` – runs on the device but is callable only from the host
  - `__device__` – runs on the device and is callable only from the device

Summary of CUDA extensions to standard C

- Variable type qualifiers
  - No type qualifier – a scalar variable will usually be stored as a register, an array will go to slow device memory. In both cases the variable is accessible only from a single thread.
  - `__shared__` – store variable in fast shared memory. Variable is accessible from all threads in the thread block.