The Graphics Processing Unit (GPU) revolution

Ramu Anandakrishnan

Virginia Tech
Invent the Future

Outline

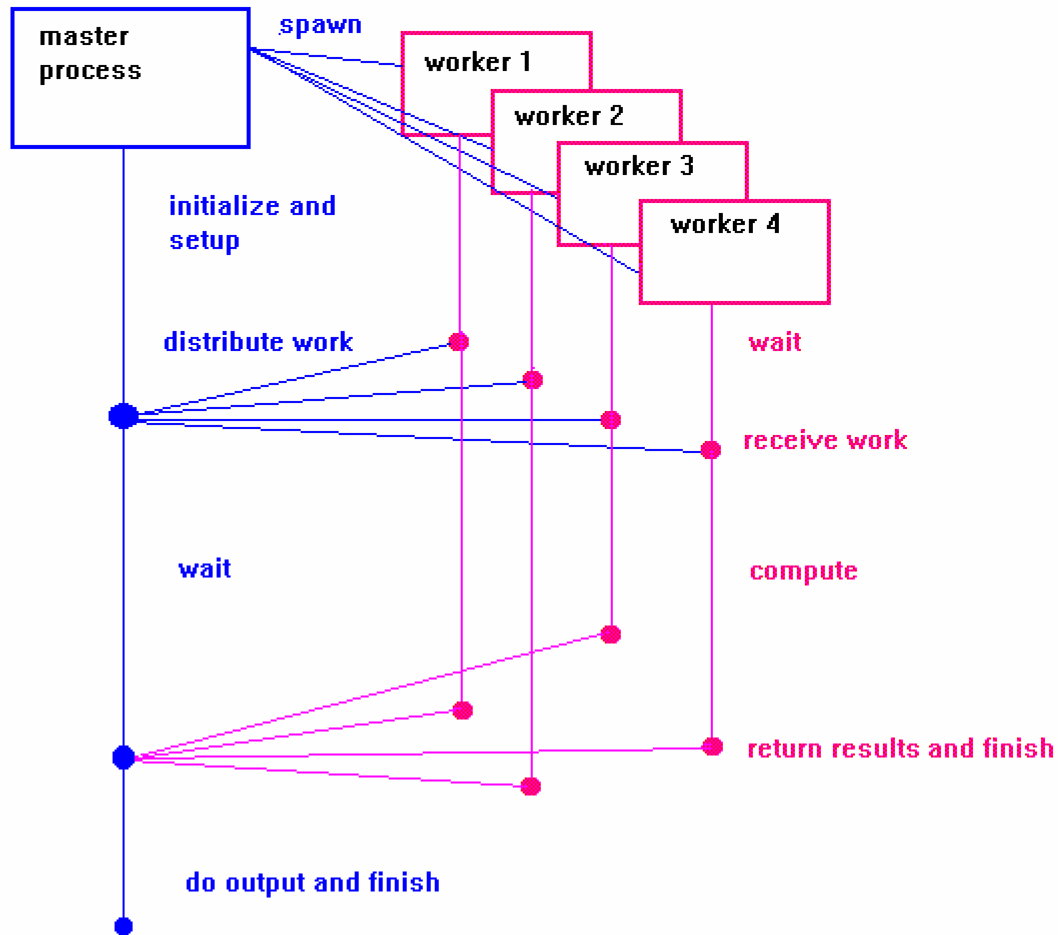
- The need for parallel processing
- Basic parallel processing concepts
- The GPU – a massively parallel processor

Lecture 1

- Introduction to GPU programming
- Overview of GPU hardware architecture
- Performance considerations
- Homework assignment

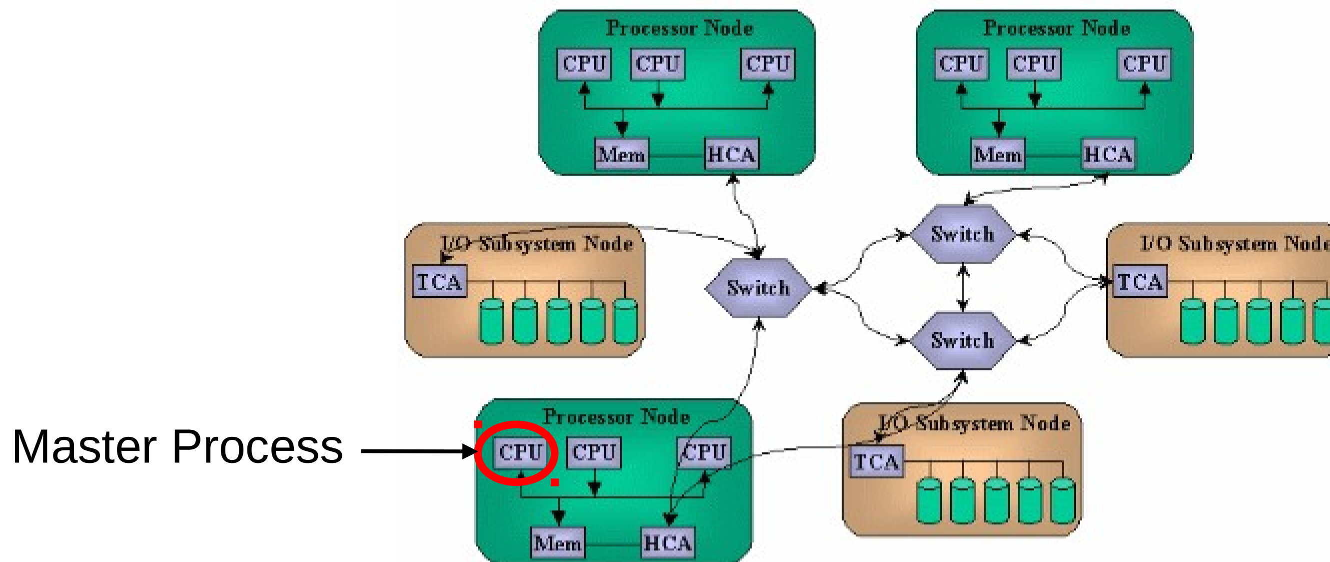
Lecture 2

Application level parallelism



A key limitation of supercomputer architecture – inter node communication

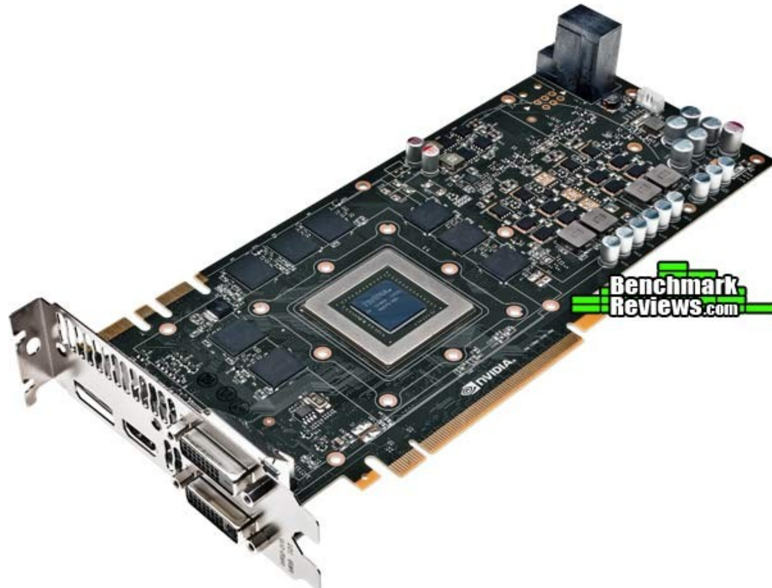
(extremetech.com)

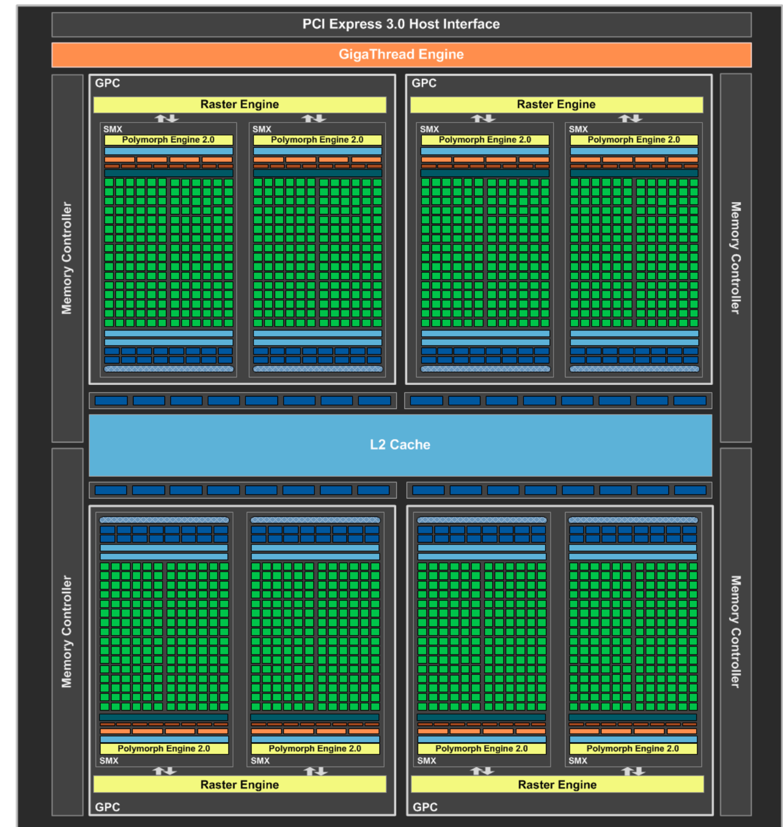


GPUs – thousands of processors on a single node

GeForce GTX 690 Specifications

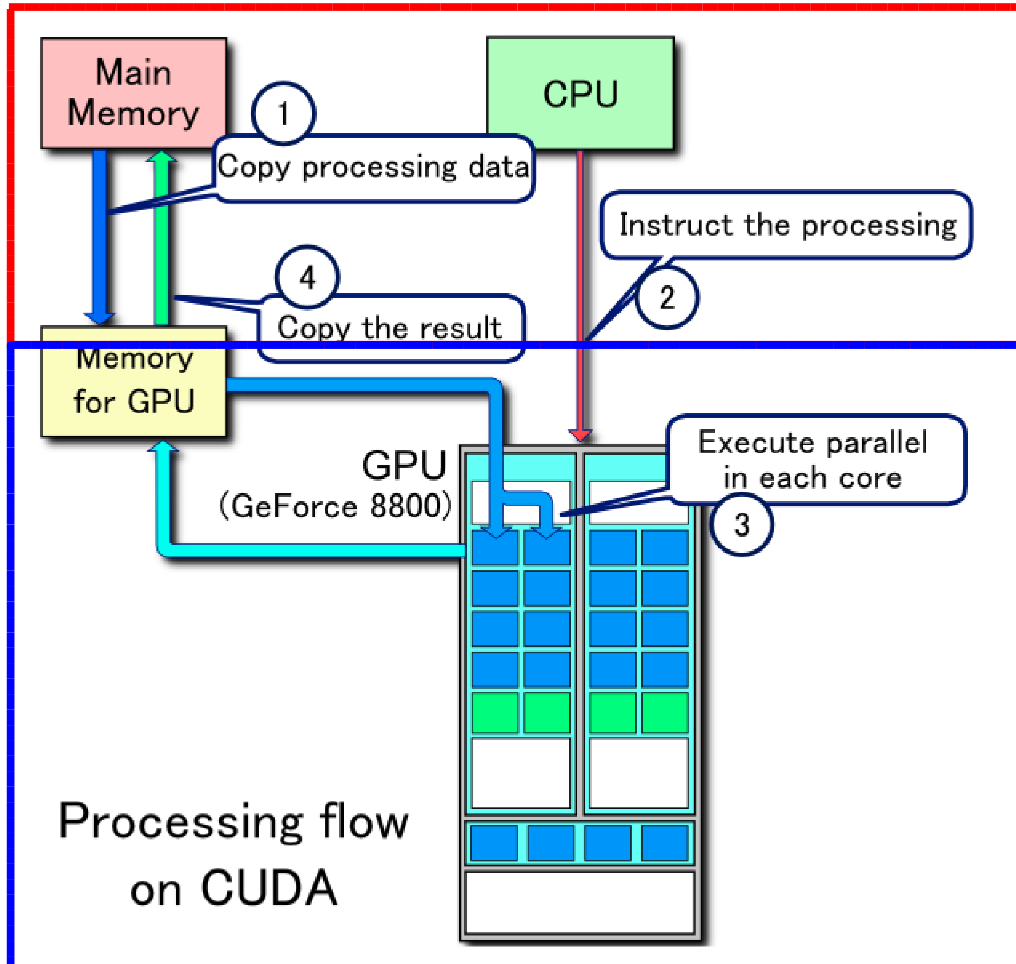
CUDA Cores	3072
Base Clock	915 MHz
Boost Clock	1019 MHz
Memory Config	4GB / 512-bit GDDR5
Memory Speed	6.0 Gbps
Power Connectors	8-pin + 8-pin
TDP	300W
Outputs	3x DL-DVI Mini-Displayport 1.2
Bus Interface	PCI Express 3.0





(www.nvidia.com)

CUDA – for running code on the GPU



http://en.wikipedia.org/wiki/CUDA

Example of CUDA process flow

1. Copy data from main mem to GPU mem
2. CPU initiates threads on the GPU
3. GPU execute parallel code
4. Copy the result from GPU mem to main mem

Example: Vector addition

Ai + Bi = Ci

A		B		C	
A0		B0		C0	Thread 0
A1		B1		C1	Thread 1
A2		B2		C2	Thread 2
. . .	+	. . .	=	. . .	
An		Bn		Cn	Thread n

1. Host: Copy data from main memory to GPU memory

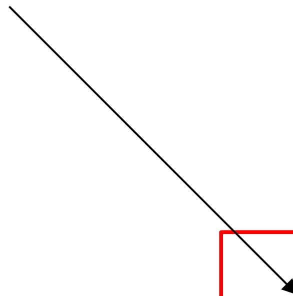

```
// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Initialize input vectors
    ...
    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    ...

}
```

2. Host: Initiate parallel threads (device kernels) on the GPU


```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    ...
}
```

```
// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Initialize input vectors
    ...
    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock – 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
    ...
}
```

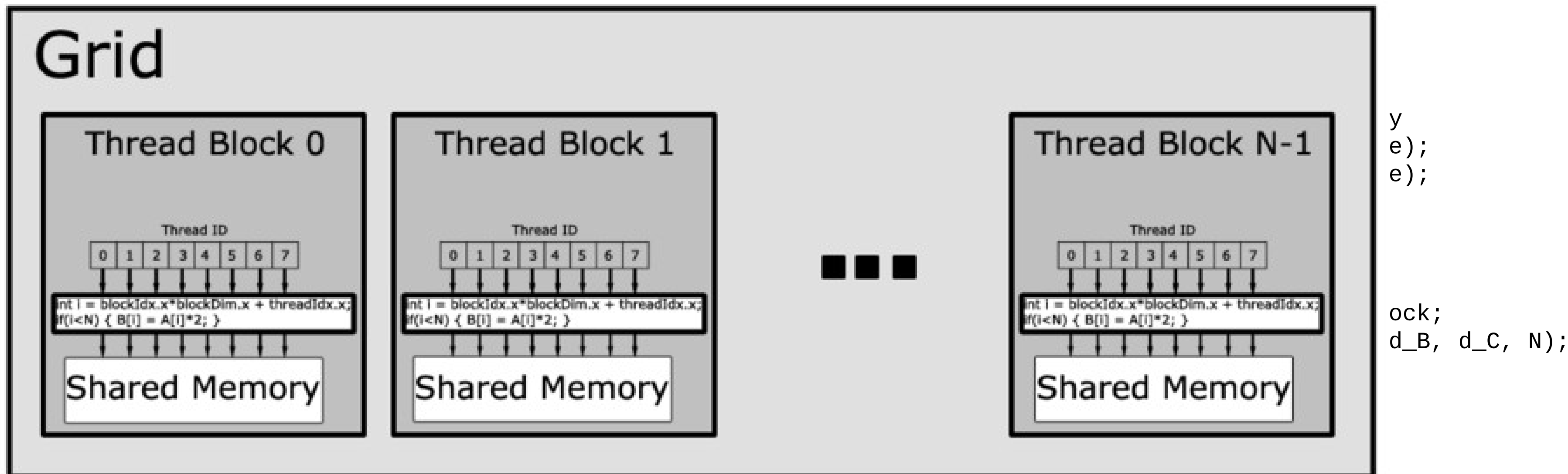
Number of threads = blocksPerGrid * threadsPerBlock



3. Device: Execute parallel code


```
// Device code
__global__ void VecAdd(float* A, float* B,
                       float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

```
// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Initialize input vectors
    ...
    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
```




```
y
e);
e);

ock;
d_B, d_C, N);
```

4. Host: Copy results from GPU memory back to main memory

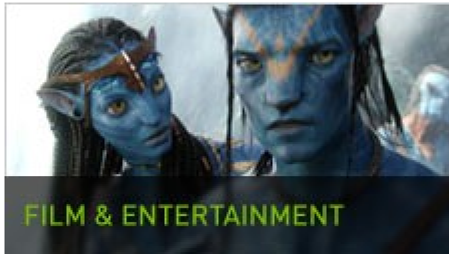

```
// Device code
__global__ void VecAdd(float* A, float* B,
                       float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

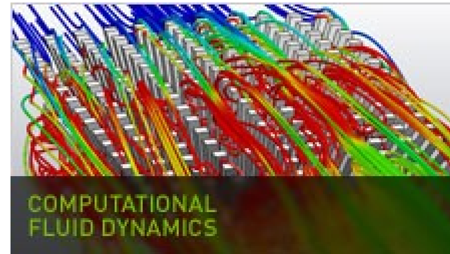
```
// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Initialize input vectors
    ...
    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock – 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    // Free host memory
    ....
}
```

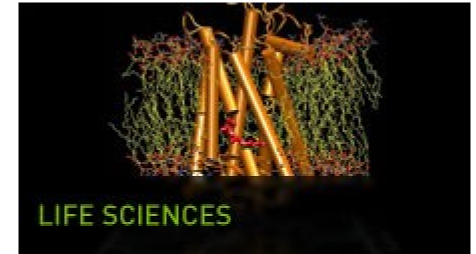
Compiling and executing GPU code

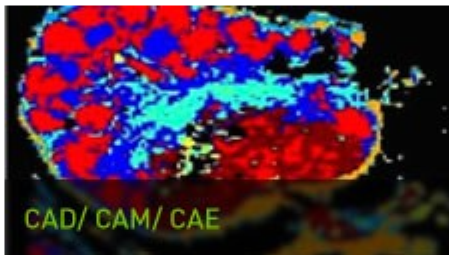
- cudaSetDevice(1) – first statement in program
- nvcc -o vectorAdd vectorAdd.cu
- ./vectorAdd

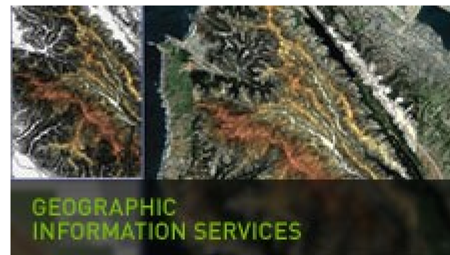
GPUs are great for highly parallelizable applications

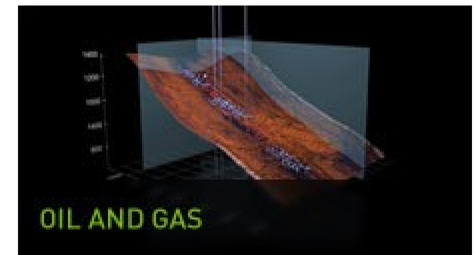






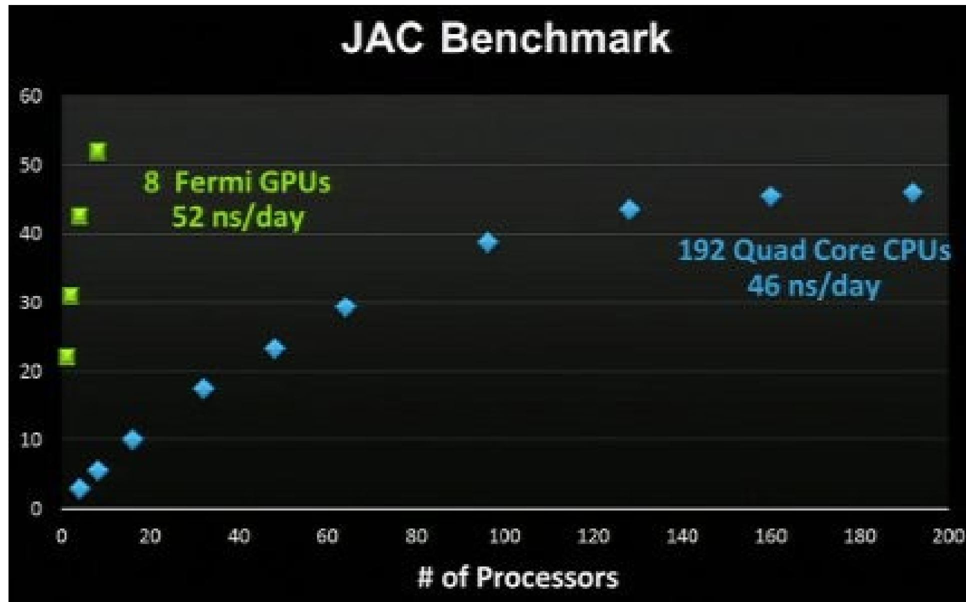






(www.nvidia.com)

For such applications, you can get the power of a supercomputer on your desktop

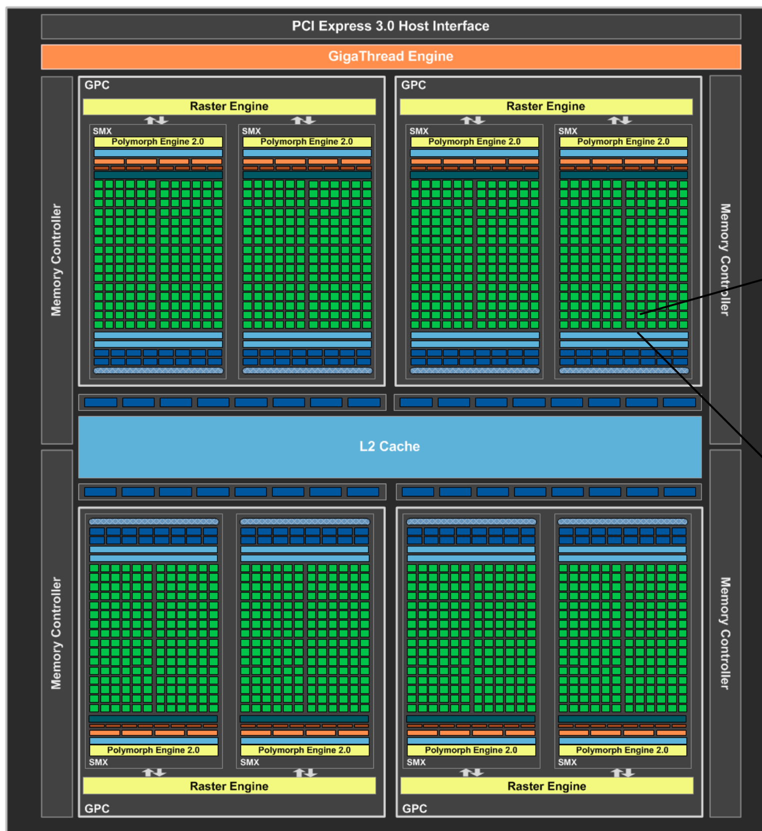


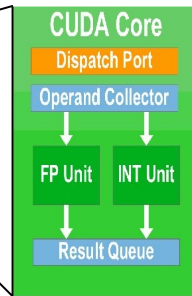
http://hexus.net/tech/news/graphics/26553-nvidia-talks-up-plans-cuda/

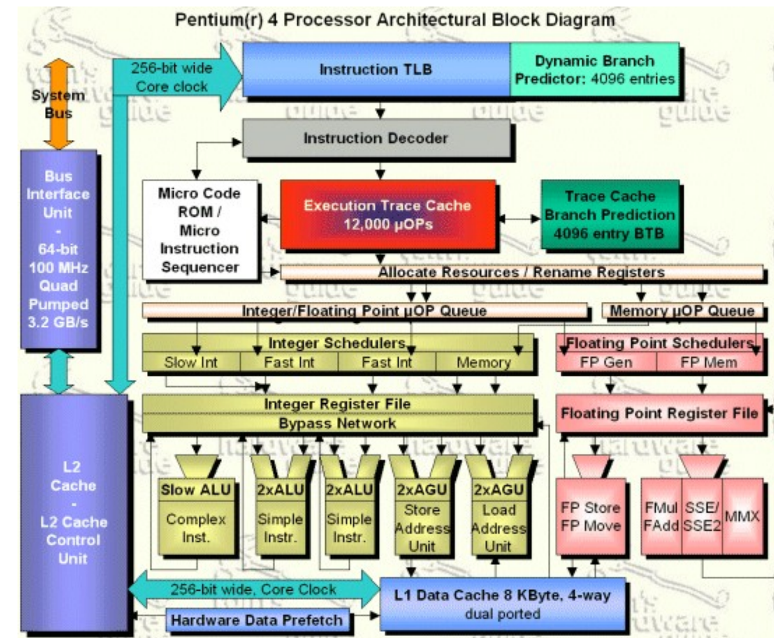
nvidia – hardware architecture

(neoseeker.com)

Tradeoffs between the CPU and GPU



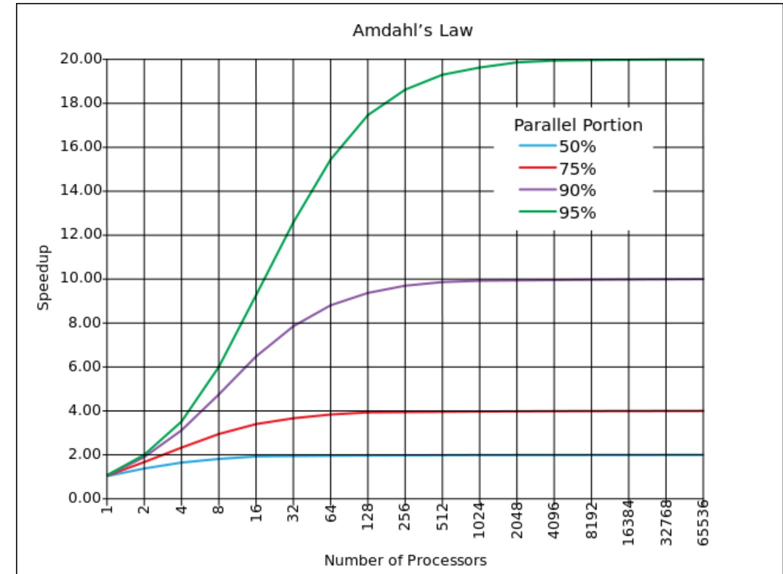




(neoseeker.com)

GPU limitations and performance considerations

- Not all code is highly parallelizable – Amdahl's law
- Severe penalty for branching (if-then-else)
- Limited GPU memory – increases code complexity
- Complex operations not available or much slower
- Communication between GPU and CPU is relatively slow
- Writing efficient code for the GPU is not easy



Homework assignment
Due date TBD

Code the Vector Addition program (http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

1. For both the CPU and GPU
2. cudaSetDevice(1) – first statement in program
3. Generate random values for input vectors
4. Output total of all Ci, i = 1 to n, for verification
5. Graphically plot execution times for different vector lengths, for CPU and GPU

Procedures for working on kuprin

1. ssh -X acct_name@bio.cs.vt.edu
2. ssh -X kuprin
3. mkdir groupxx
4. cd groupxx
5. Use vi, gvim, ed, or gedit, to create/edit your program
6. cudaSetDevice(1) – first statement in program

Ethical use of shared computer resources

- Do not access accounts and directories that are not yours
- Use resources only for the authorized purpose – assignments and projects
- Do not allow anyone else to use your account
- Do not try anything that might crash the machine
- Do not overload the machine so that others can not use it (denial of service)
- Follow all licensing and copyright laws
- Report any violations of these rules

If you’re interested in learning more

- Yong Cao
  - CS4204 "Computer Graphics"
  - CS4644 "Creative Computing Studio: Video Game Design"
  - CS5984 "Advanced Computer Graphics: Parallel Computing and Visualization on GPU"
  - CS6204 “Character Animation"
- Wu Feng – has several GPU project, led the HokieSpeed project
- Alexey Onufriev – we have some projects
- CS summer seminar
- http://developer.amd.com/pages/default.aspx
- http://developer.nvidia.com/category/zone/cuda-zone
- HokieSpeed