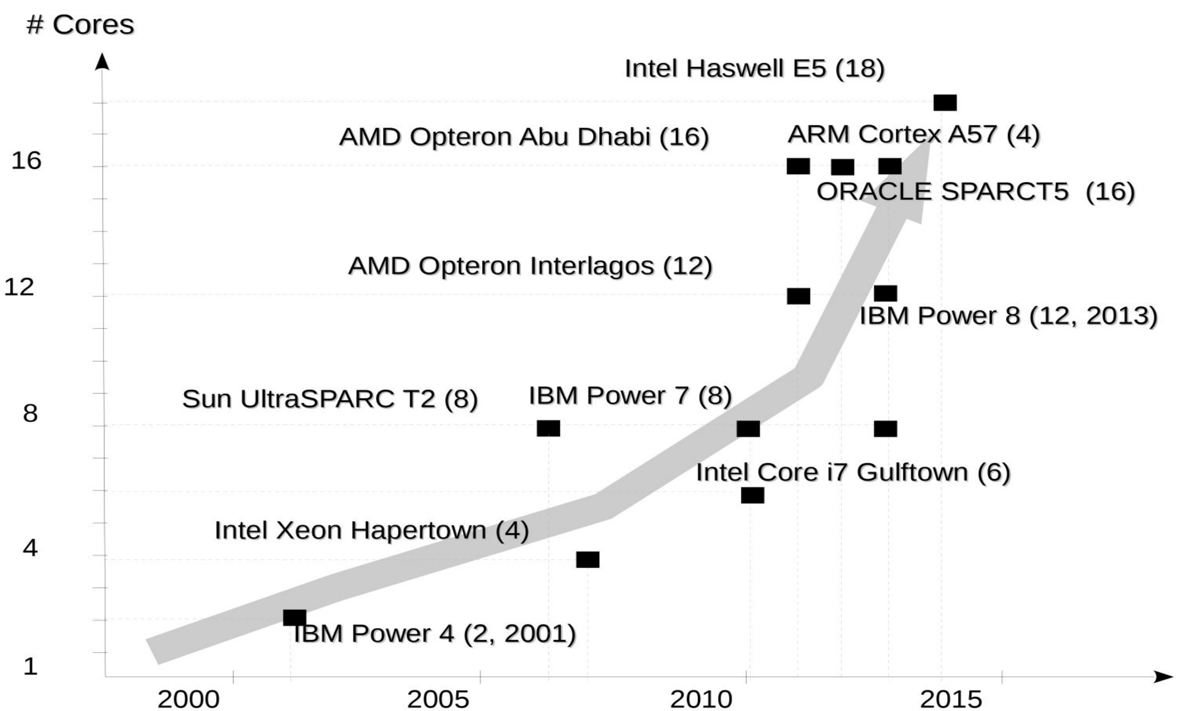
Parallel & Distributed Computing
Lecture Week 11/2:
GPU Based Computing

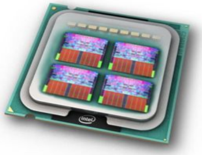
Farhad M. Riaz

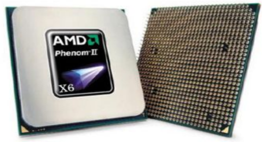
Farhad.Muhammad@numl.edu.pk

Department of Computer Science
NUML, Islamabad

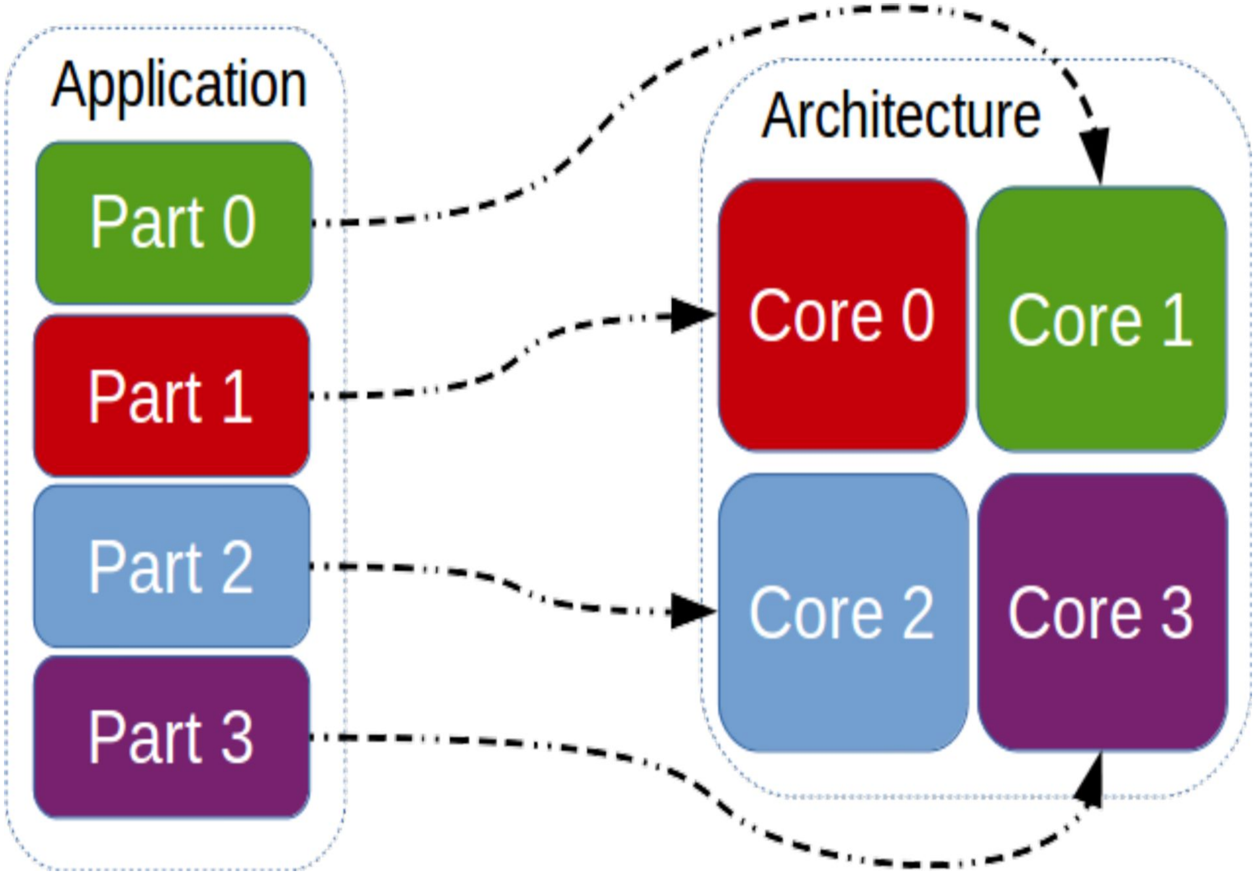
Review: Multicore Trend



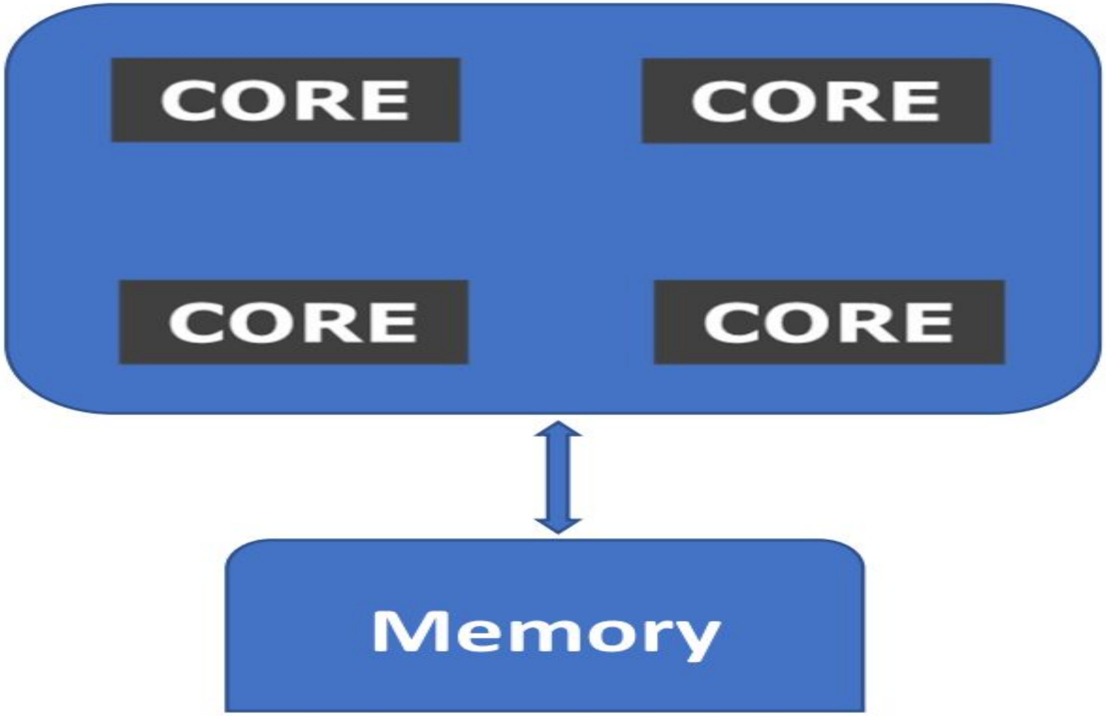




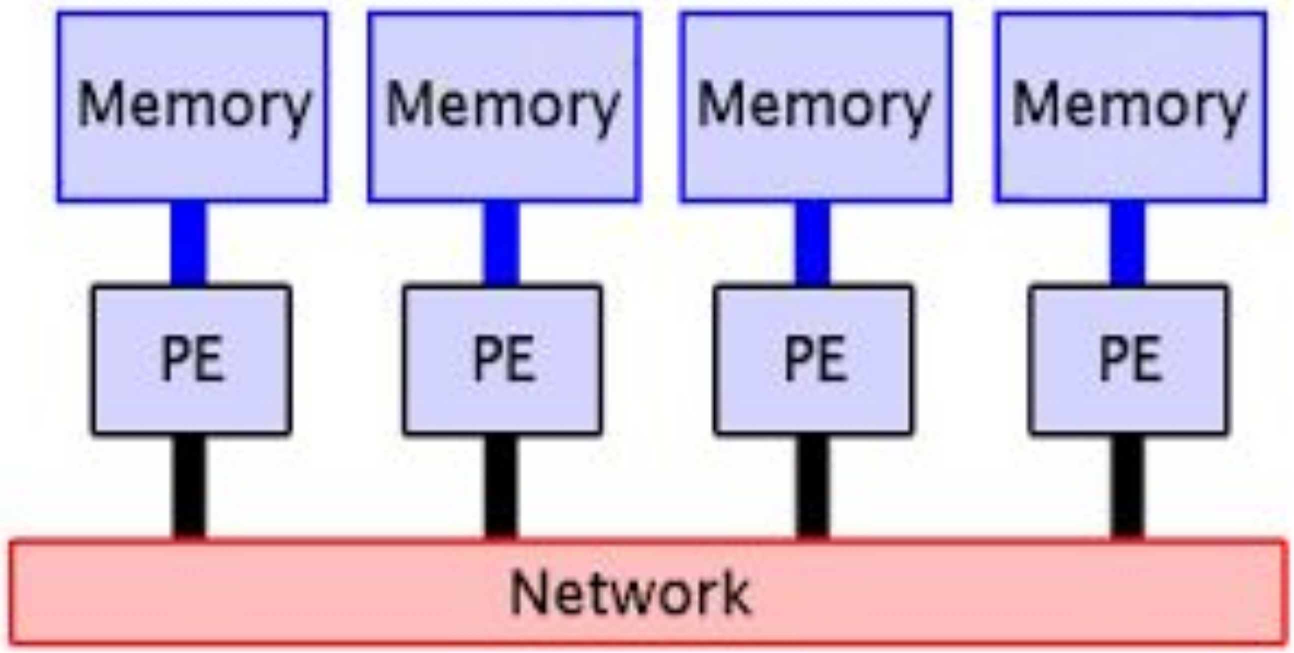
Review: Parallel Processing



Review: Shared Memory Model



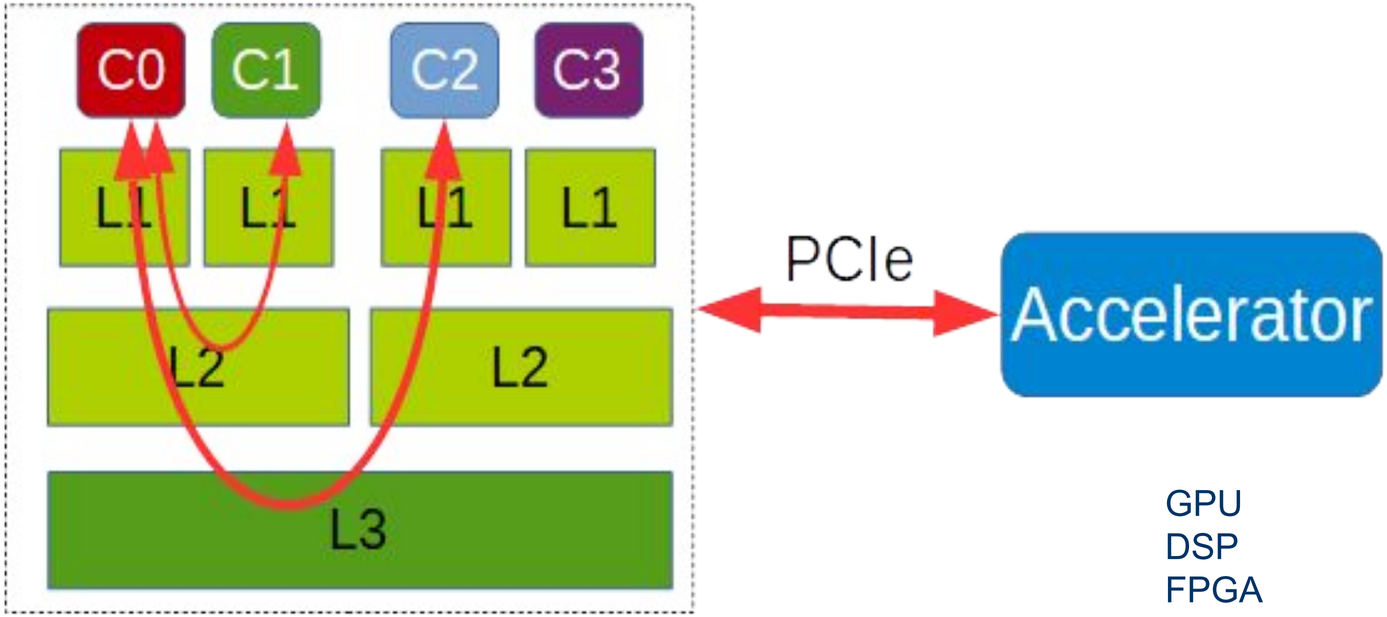
Review: Distributed Memory Systems



Accelerator

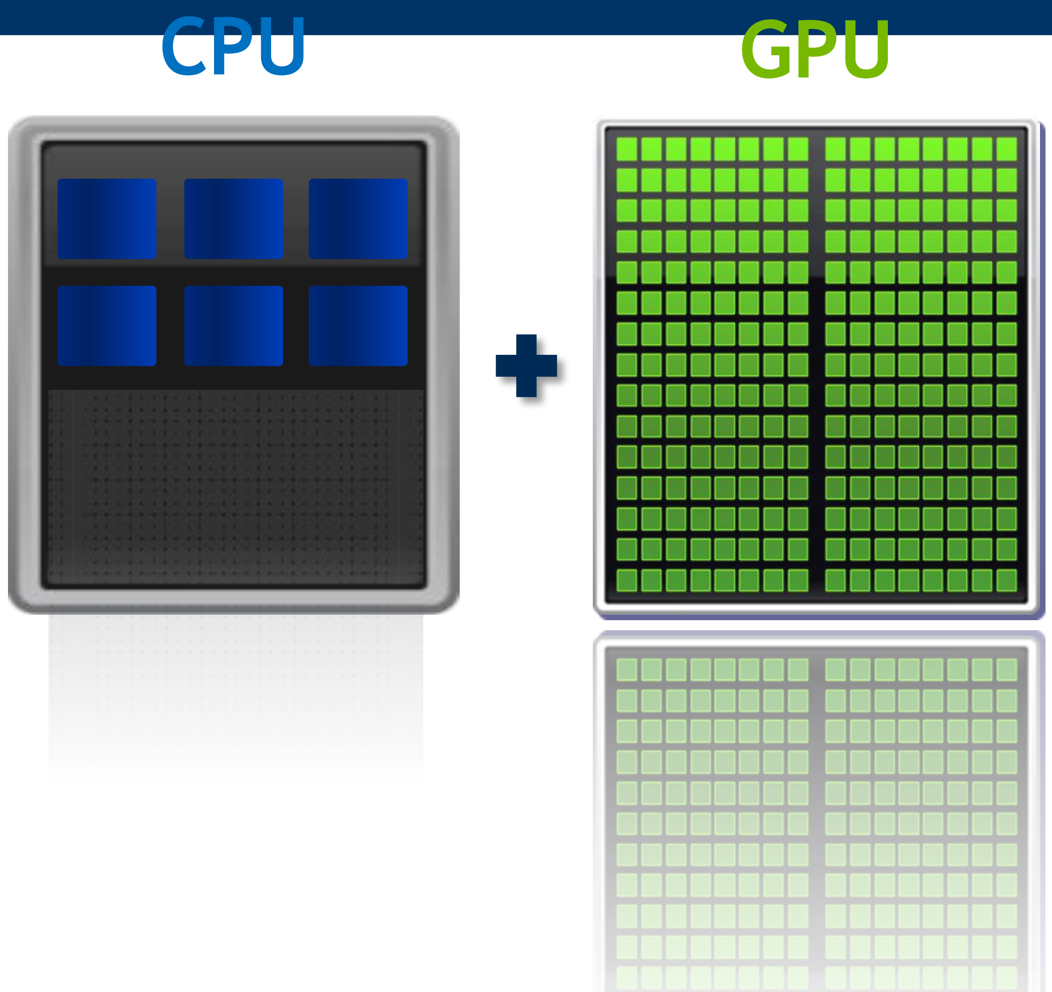
- An **accelerator** is a hardware device or a software program with a main function of enhancing the overall performance of the **computer**.

Accelerator-based computing

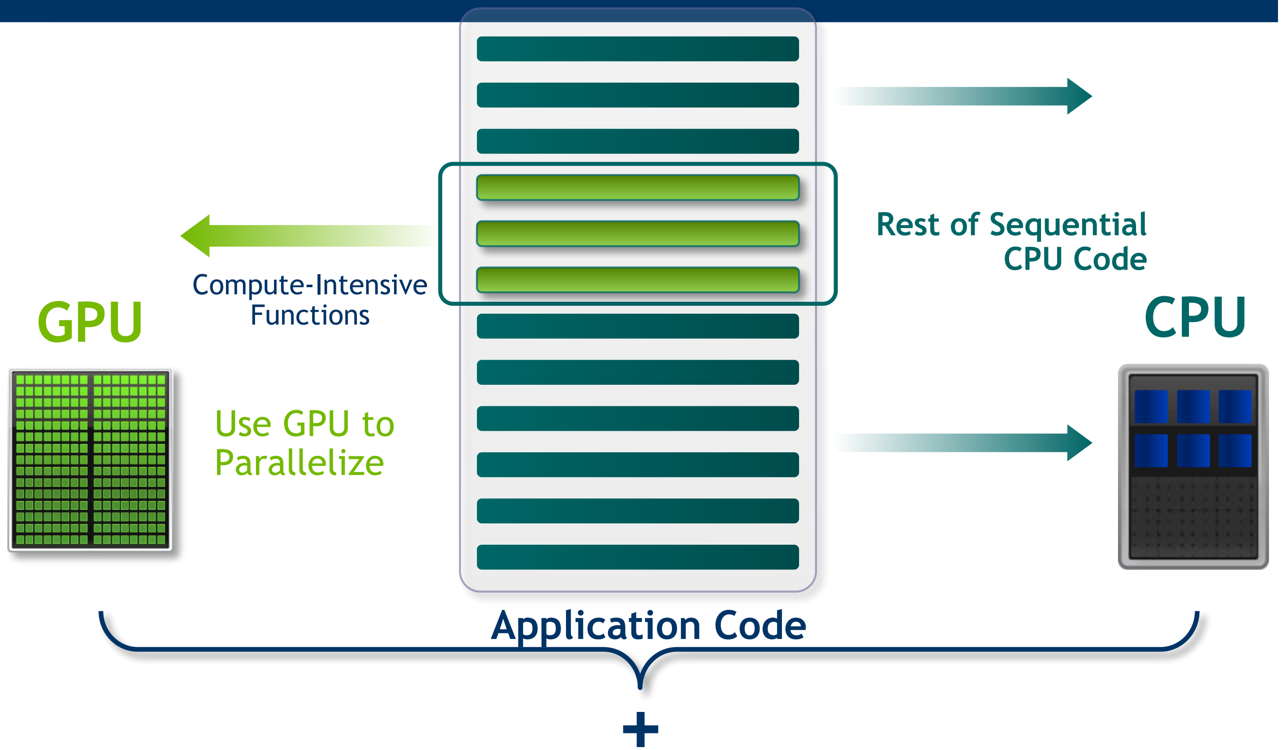


GPU
DSP
FPGA
QPU
TPU

Add GPUs: Accelerate Science Applications

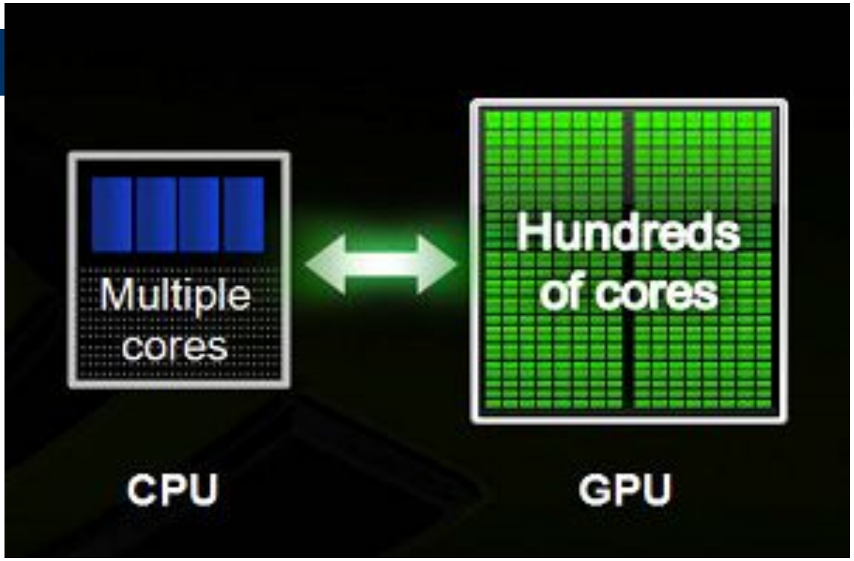


Small Changes, Big Speed-up



CPU vs GPU

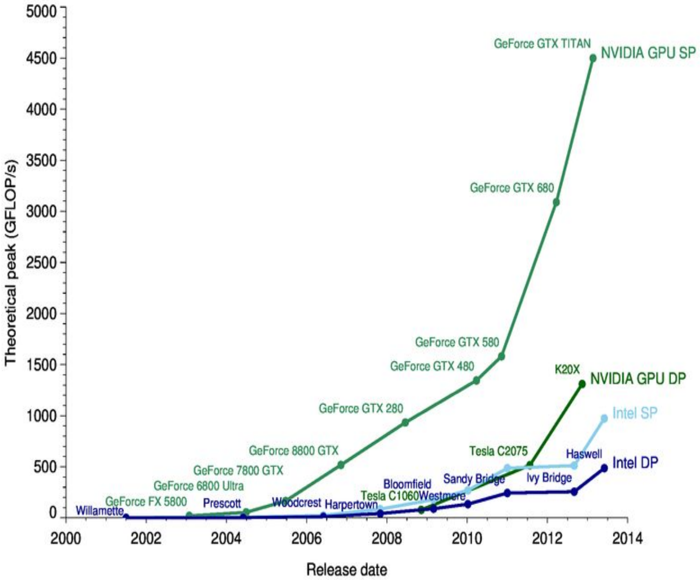
- GPU has higher parallelism than CPU
- CPU has better serial processing capabilities
- CPU-GPU comprise a heterogeneous system



- Best performance is using both CPU & GPU

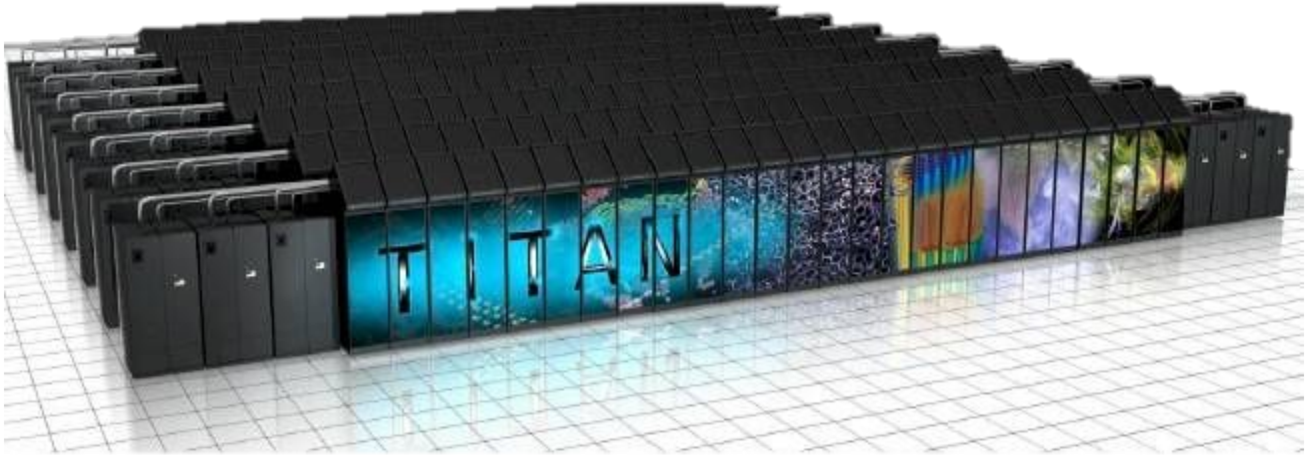
CPU vs GPU

- GPU peak performance much higher than CPU
- Only achievable for highly paralle applications
  - Graphics
  - Scientific
  - Many others
- Made possible by many small GPU cores



CPU vs GPU

- Many of the Top 10 supercomputers use GPUs

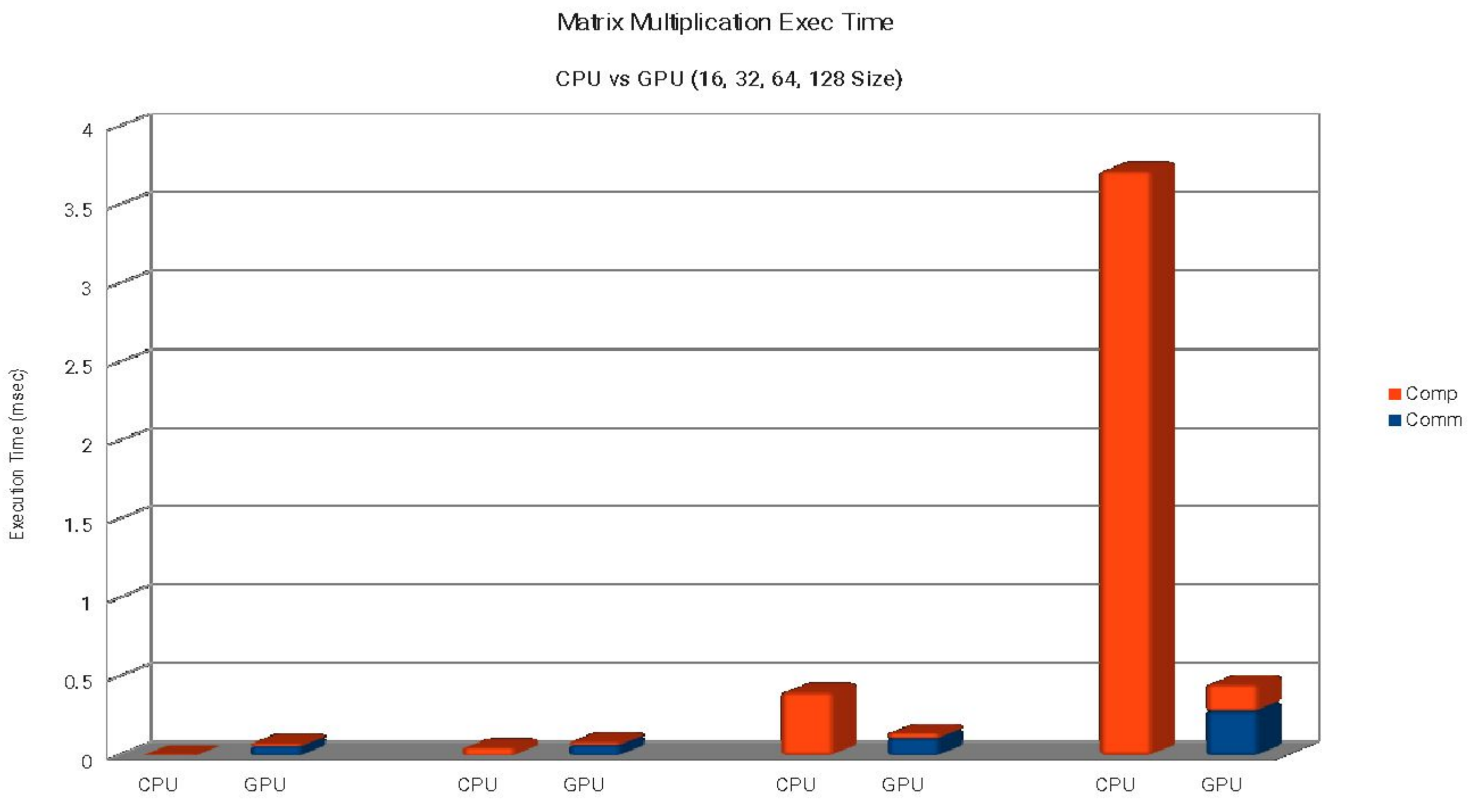


Titan supercomputer

Vector Add Example: Timings

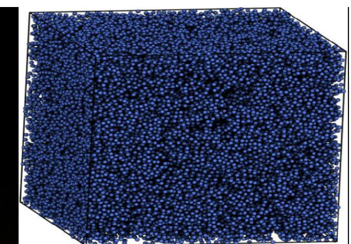
- CPU : 93 msec
- GPU (CUDA) : 51 msec (4 + 47)

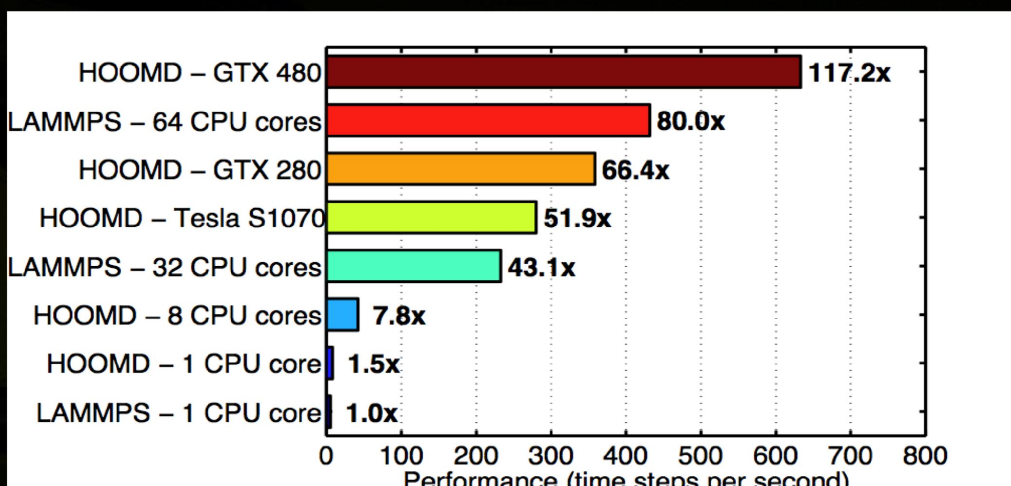
Matrix Multiplication Results



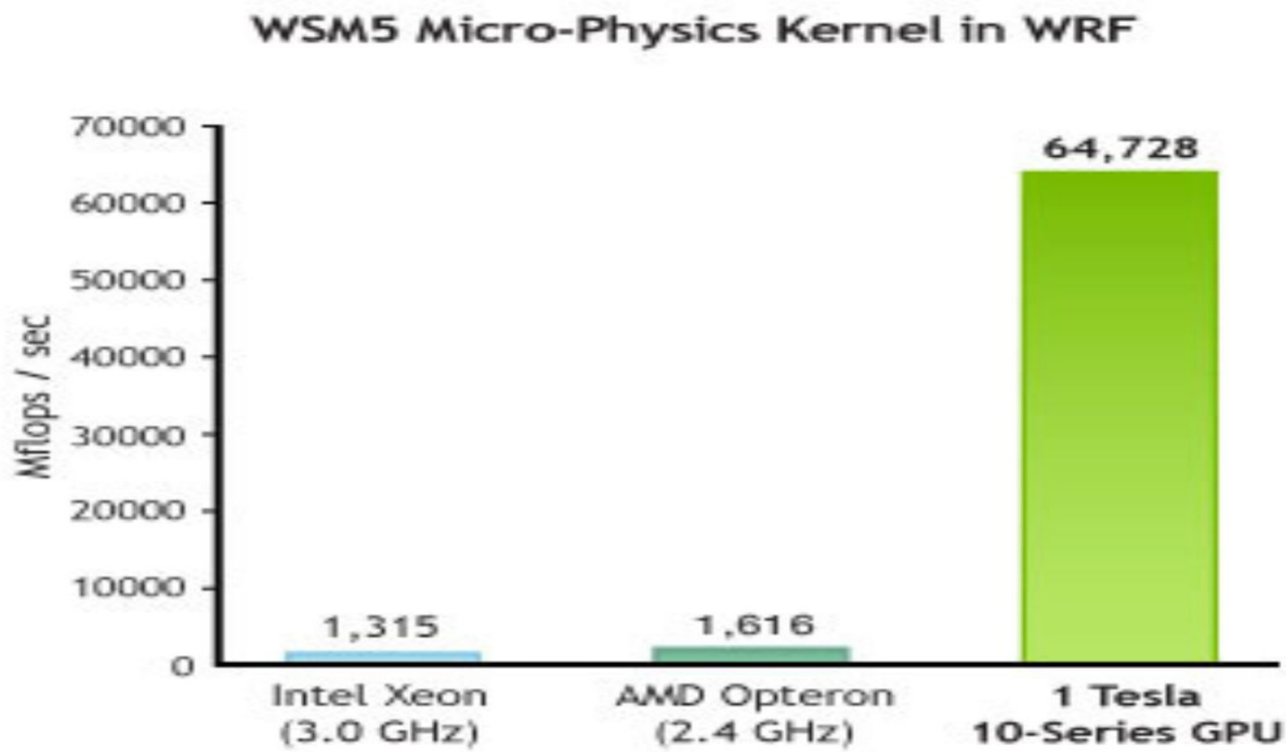
HOOMD-blue Benchmark

- 64,000 particle Lennard-Jones fluid simulation
- representative of typical performance gains





*CPU: Intel Xeon E5540 @ 2.53GHz


WSM5 Micro-Physics Kernel in WRF
Mflops / sec
70000
60000
50000
40000
30000
20000
10000
0
1,315
1,616
64,728
Intel Xeon
(3.0 GHz)
AMD Opteron
(2.4 GHz)
1 Tesla
10-Series GPU

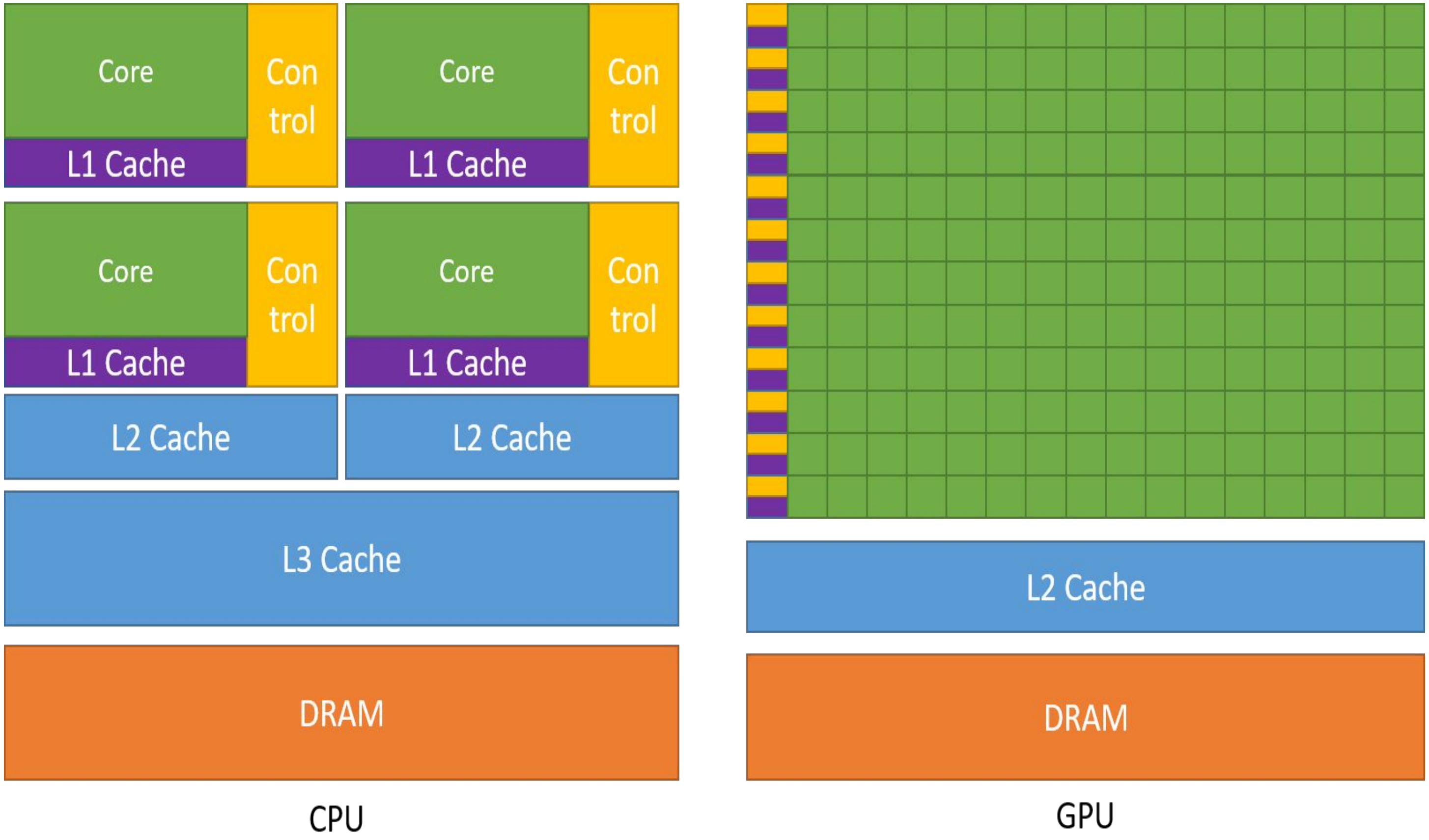
The Benefits of Using GPUs

- The Graphics Processing Unit (GPU) provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope.
- Many applications leverage these higher capabilities to run faster on the GPU than on the CPU .
- Other computing devices, like FPGAs, are also very energy efficient, but offer much less programming flexibility than GPUs.

The Benefits of Using GPUs

- The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control.

Chip resources for a CPU versus a GPU.



CUDA: A General-Purpose Parallel Computing Platform and Programming Model

- In November 2006, NVIDIA® introduced CUDA®, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

- CUDA comes with a software environment that allows developers to use C++ as a high-level programming language.

GPU Computing Applications

Libraries and Middleware

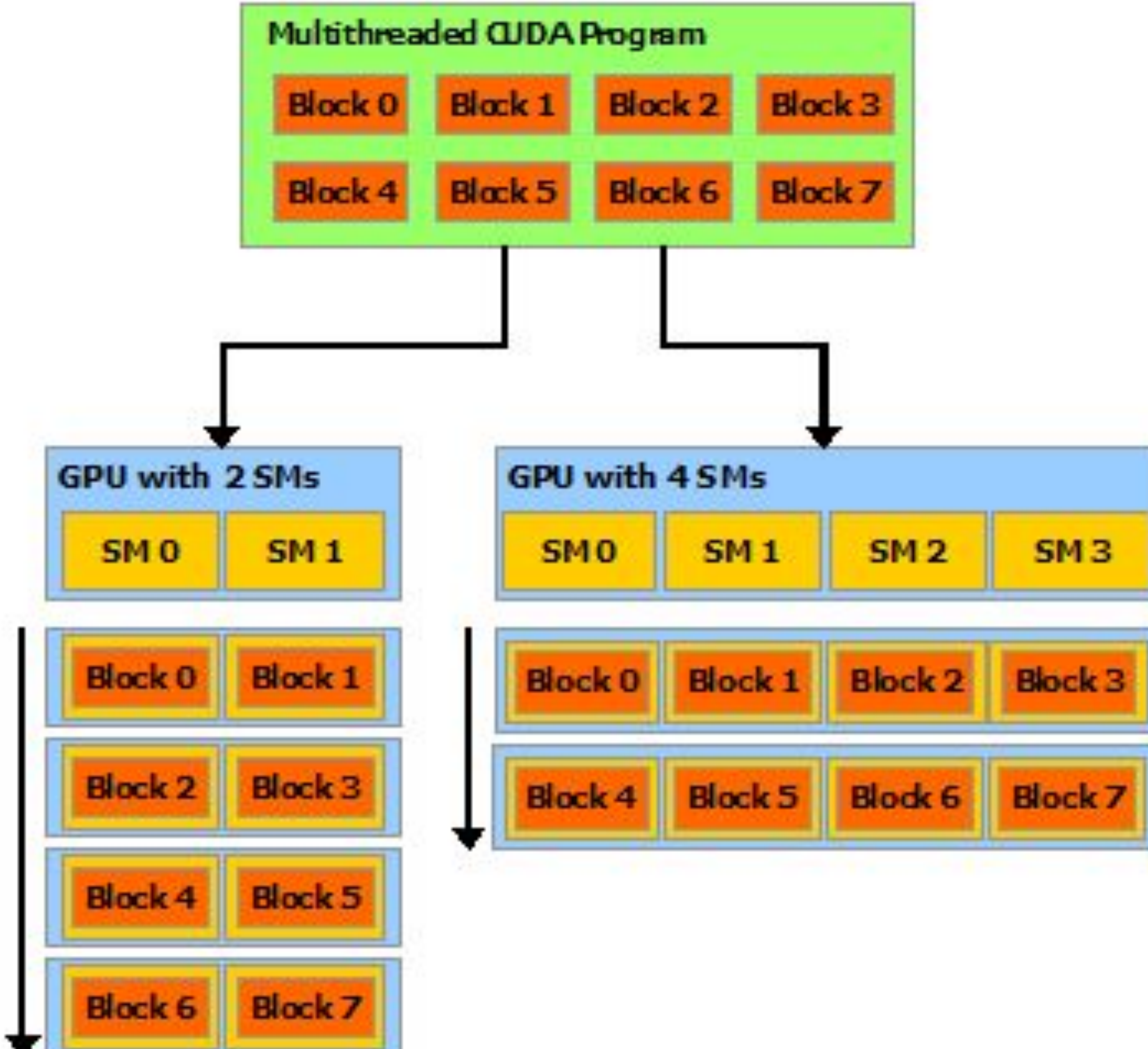
cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series



Multithreaded CUDA Program
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 2 SMs
SM 0
SM 1
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 4 SMs
SM 0
SM 1
SM 2
SM 3
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7

Kernal Definition

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
int i = threadIdx.x; C[i] = A[i] + B[i];
}
int main()
{ ... // Kernel invocation with N threads
VecAdd<<<1, N>>>(A, B, C); ...
}
```

```
// Kernel definition
 __global__ void MatAdd(float A[N][N], float B[N][N], float
C[N][N]) {
int i = threadIdx.x; int j = threadIdx.y; C[i][j] = A[i][j] + B[i][j];
 }
 int main()
 { ... // Kernel invocation with one block of N * N * 1 threads
int numBlocks = 1;
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C); ...
}
```

3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC Directives

Programmin
g
Languages

"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

3 Ways to Accelerate Applications

Applications

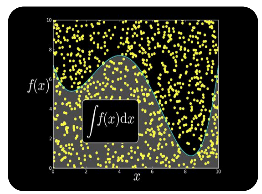
Libraries	OpenACC Directives	Programming Languages
“Drop-in” Acceleration	Easily Accelerate Applications	Maximum Flexibility

Libraries: Easy, High-Quality Acceleration

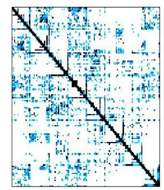
- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications
- **Performance:** NVIDIA libraries are tuned by experts

Some GPU-accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

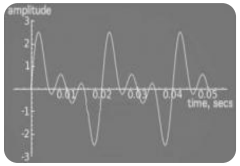
Vector Signal Image Processing

CULA | tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and Multicor…



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

CUSP

Sparse Linear Algebra

Thrust

C++ STL Features for CUDA

3 Steps to CUDA-accelerated application

- **Step 1:** Substitute library calls with equivalent CUDA library calls

  `saxpy ( … )` `cublasSaxpy ( … )`

- **Step 2:** Manage data locality

  - with CUDA: `cudaMalloc(), cudaMemcpy(), etc.`
  - with CUBLAS: `cublasAlloc(), cublasSetVector(), etc.`

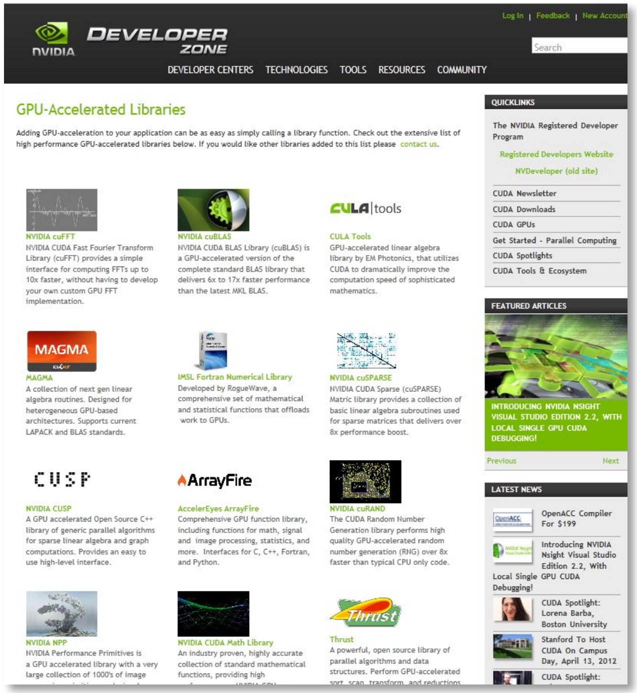
- **Step 3:** Rebuild and link the CUDA-accelerated library

  `nvcc myobj.o –l cublas`

Explore the CUDA (Libraries) Ecosystem

- CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone:

  developer.nvidia.com/cuda-tools-ecosystem

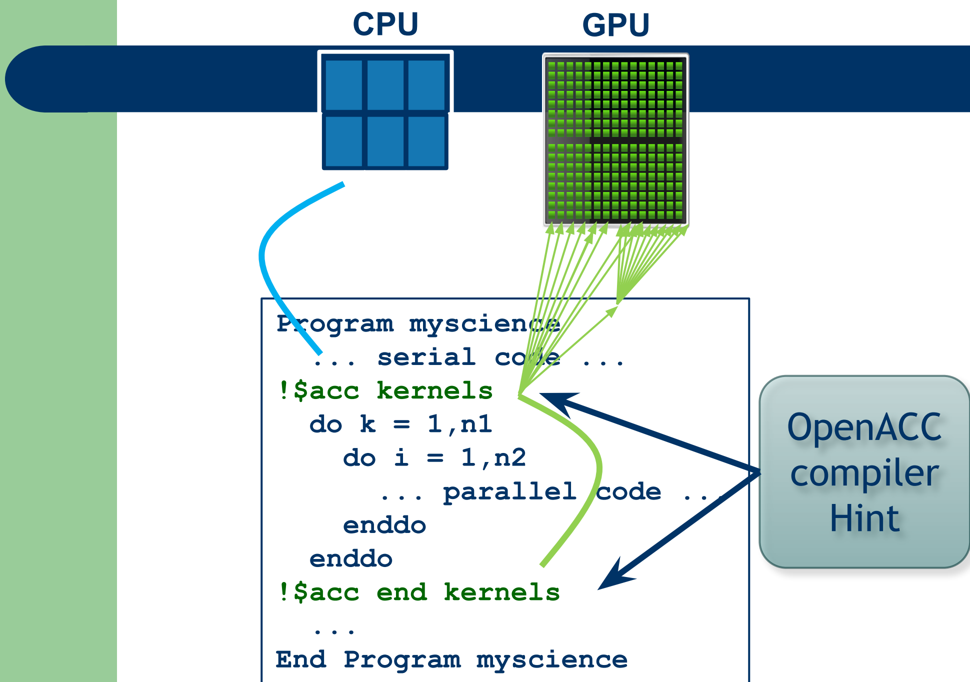


3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

OpenACC Directives



Your original Fortran or C code

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

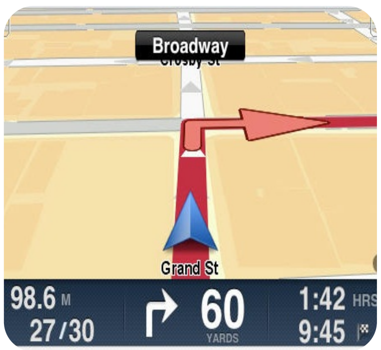
OpenACC
The Standard for GPU Directives

- **Easy:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU

Directives: Easy & Powerful

Real-Time Object Detection

Global Manufacturer of Navigation Systems



5x in 40 Hours

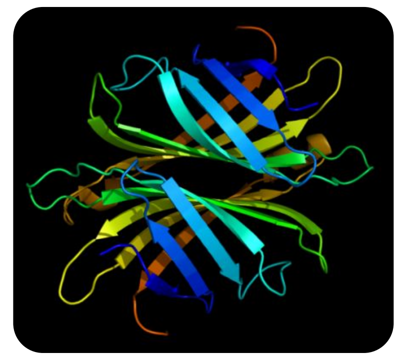
Valuation of Stock Portfolios using Monte Carlo

Global Technology Consulting Company

2x in 4 Hours

Interaction of Solvents and Biomolecules

University of Texas at San Antonio



5x in 8 Hours

“ ”

Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications.

-- Developer at the Global Manufacturer of Navigation Systems

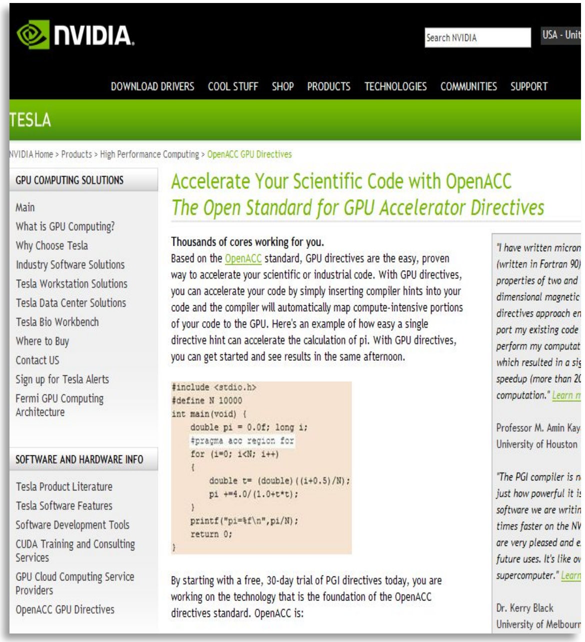
Start Now with OpenACC Directives

Sign up for a free trial of the directive's compiler now!

Free trial license to PGI Accelerator

Tools for quick ramp

www.nvidia.com/gpudirectives



3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

GPU Programming Languages

Numerical analytics	
Fortran	OpenACC, CUDA Fortran
C	OpenACC, CUDA C
C++	Thrust, CUDA C++
Python	PyCUDA, Copperhead
F#	Alea.cuBase

Learn More

These languages are supported on all CUDA-capable GPUs. [redacted] or desktop PC!

CUDA C/C++
http://developer.nvidia.com/cuda-toolkit

Thrust C++ Template Library
http://developer.nvidia.com/thrust

CUDA Fortran
http://developer.nvidia.com/cuda-toolkit

PyCUDA (Python)
http://mathema.tician.de/software/pycuda

GPU.NET
http://tidepowerd.com

MATLAB
http://www.mathworks.com/discovery/matlab-gpu.html

Mathematica
http://www.wolfram.com/mathematica/new-in-8/cuda-and-opencl-support/

Getting Started

- Nsight IDE (Eclipse or Visual Studio): www.nvidia.com/nsight
- Programming Guide/Best Practices:
  - docs.nvidia.com
- Questions:
  - NVIDIA Developer forums: devtalk.nvidia.com
  - Search or ask on: www.stackoverflow.com/tags/cuda
- General: www.nvidia.com/cudazone

Summary

- Accelerator based computing is trending
- GPU is a famous example of an accelerator
- GPUs can speedup applications by order of magnitude

Example

- Given two vectors (i.e. arrays), we would like to add them together in a third array.
- For example: A = {0, 2, 4, 6, 8}
- B = {1, 1, 2, 2, 1}
- Then A + B = C = {1, 3, 6, 8, 9}
- The array is 5 elements long, so our approach will be to create 5 different threads.
- The first thread is responsible for computing C[0] = A[0] + B[0].
- The second thread is responsible for computing C[1] = A[1] + B[1],

```c
#include "stdio.h"

#define N 10

void add(int *a, int *b, int *c)
{
        int tID = 0;
        while (tID < N)
        {
                c[tID] = a[tID] + b[tID];
                tID += 1;
        }
}
```

```
int main()
{
        int a[N], b[N], c[N];
        // Fill Arrays
        for (int i = 0; i < N; i++)
        {
                a[i] = i,
                b[i] = 1;
        }
        add (a, b, c);
        for (int i = 0; i < N; i++)
        {
                printf("%d + %d = %d\n", a[i], b[i], c[i]);
        }
        return 0;
}
```

```
#include "stdio.h"
#define N 10

__global__ void add(int *a, int *b, int *c)
{
        int tID = blockIdx.x;
        if (tID < N)
        {
                c[tID] = a[tID] + b[tID];
        }
}
```

```
int main()
{
        int a[N], b[N], c[N];
        int *dev_a, *dev_b, *dev_c;

        cudaMalloc((void **) &dev_a, N*sizeof(int));
        cudaMalloc((void **) &dev_b, N*sizeof(int));
        cudaMalloc((void **) &dev_c, N*sizeof(int));

        // Fill Arrays
        for (int i = 0; i < N; i++)
        {
                a[i] = i,
                b[i] = 1;
        }
```

```
    cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);

    add<<<N,1>>>(dev_a, dev_b, dev_c);

    cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);

    for (int i = 0; i < N; i++)
    {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }
    return 0;
}
```

That's all for today!!