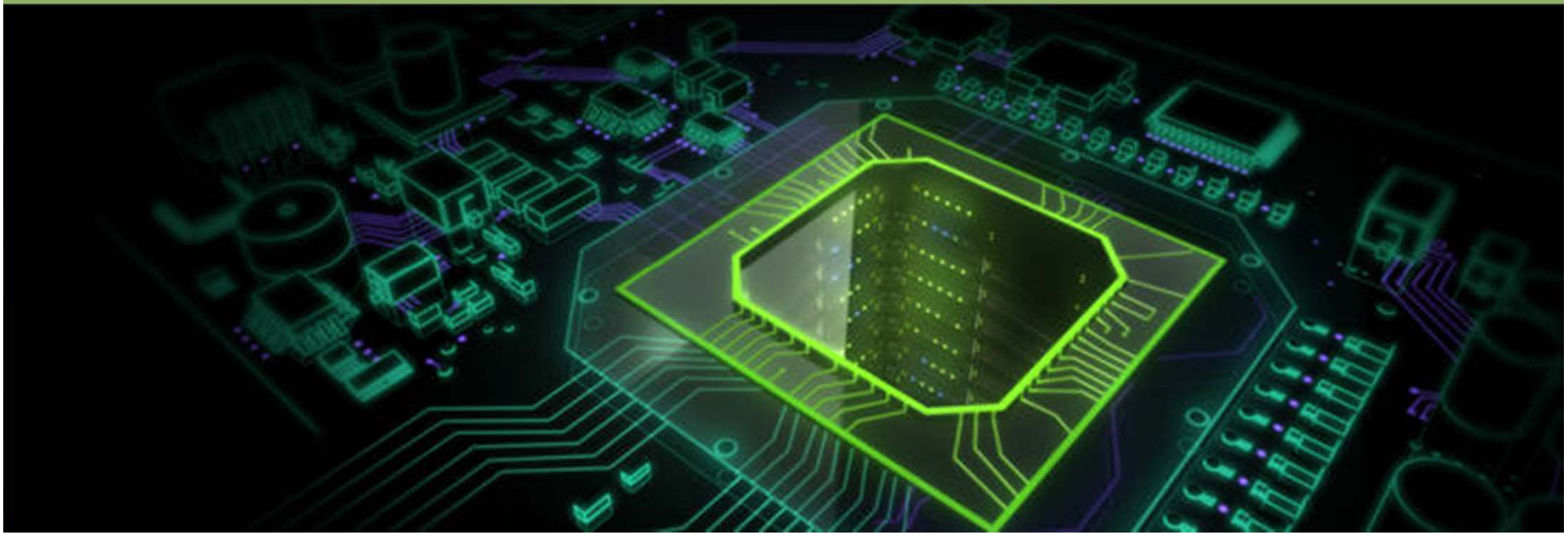
Parallel Programming:

An Introduction to GPU Computing

Will Cunningham



Outline

1. Motivation
2. Hardware Architecture
3. CUDA Programming
4. Connection to Physics
5. Putting it All Together: The Sparse Conjugate Gradient Method

What is a GPU?

- A GPU is a Graphics Processing Unit
- It is optimized for rendering a 2D matrix of many, many pixels all at the same time
- The GeForce 256 contains 22 million transistors compared to 9 million in the Pentium 3 CPU chip
- Despite a slower clock, GPUs can have hundreds of cores, so hybrid systems have become very popular over the past decade

Who Cares?

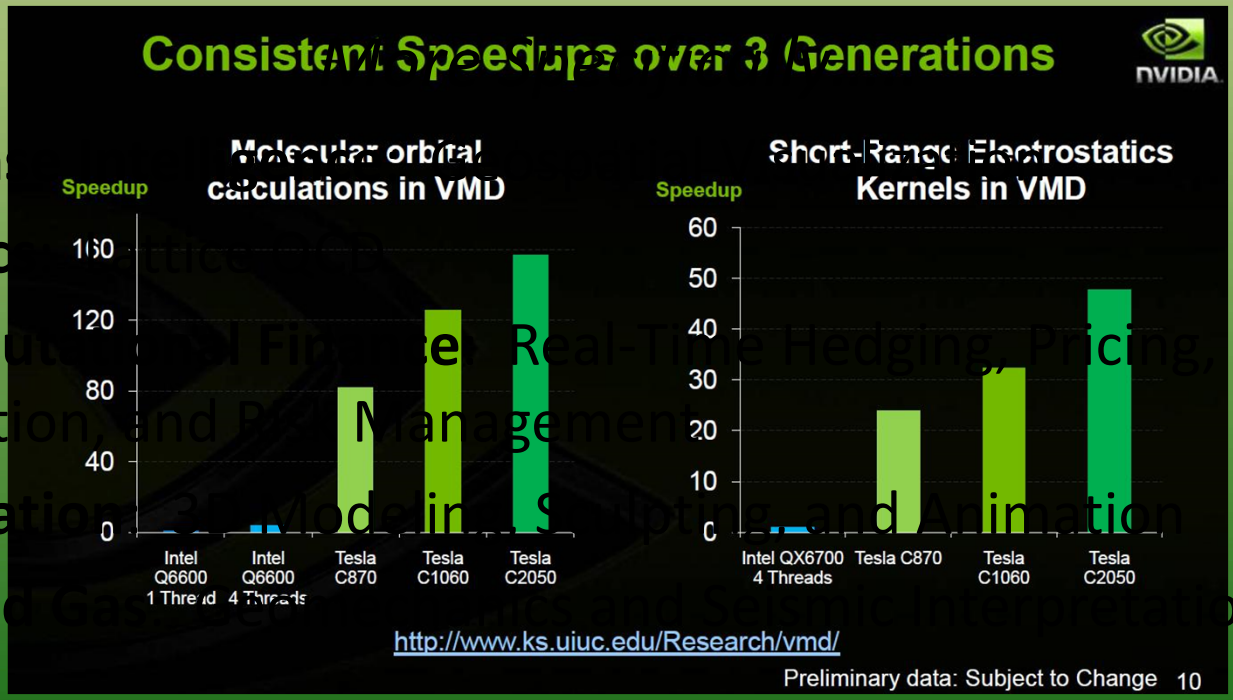
"GPUs have evolved to the point where many real-world applications are easily implemented on them and run significantly faster than on multi-core systems. Future computing architectures will be hybrid systems with parallel-core GPUs working in tandem with multi-core CPUs."

-Prof. Jack Dongarra
Director of the Innovative Computing Laboratory
University of Tennessee

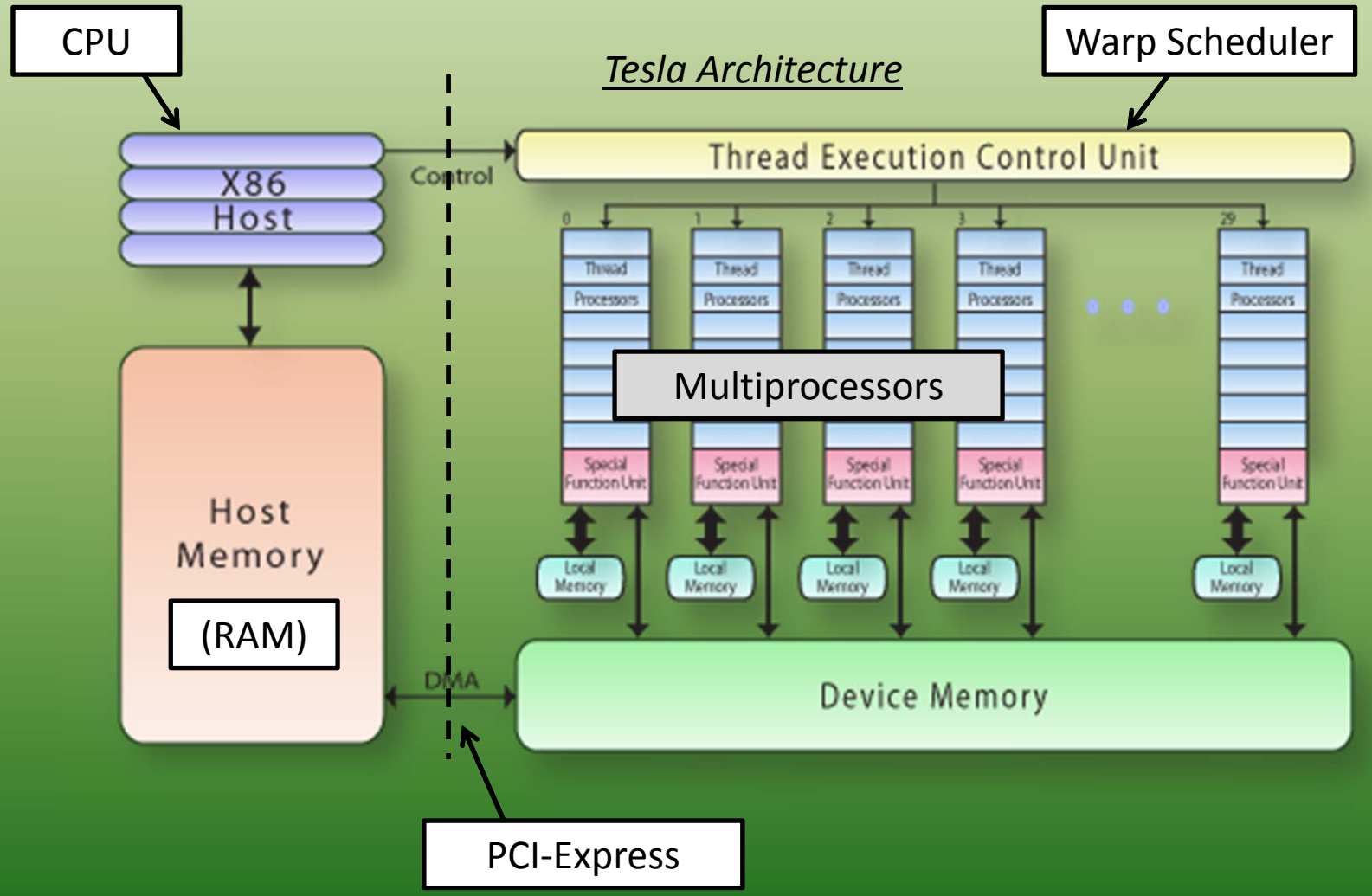
Real-World Applications

There are packages for bioinformatics, molecular dynamics, quantum chemistry, materials science, visualization/docking software, numerical analytics, physics, weather/climate forecasting, defense intelligence, computational finance, CAD, fluid dynamics, structural mechanics, electronic design automation, animation, graphics, and seismic processing.

- **Defen**
- **Physic**
- **Compu** Valuation, and R
- **Anima**
- **Oil and**

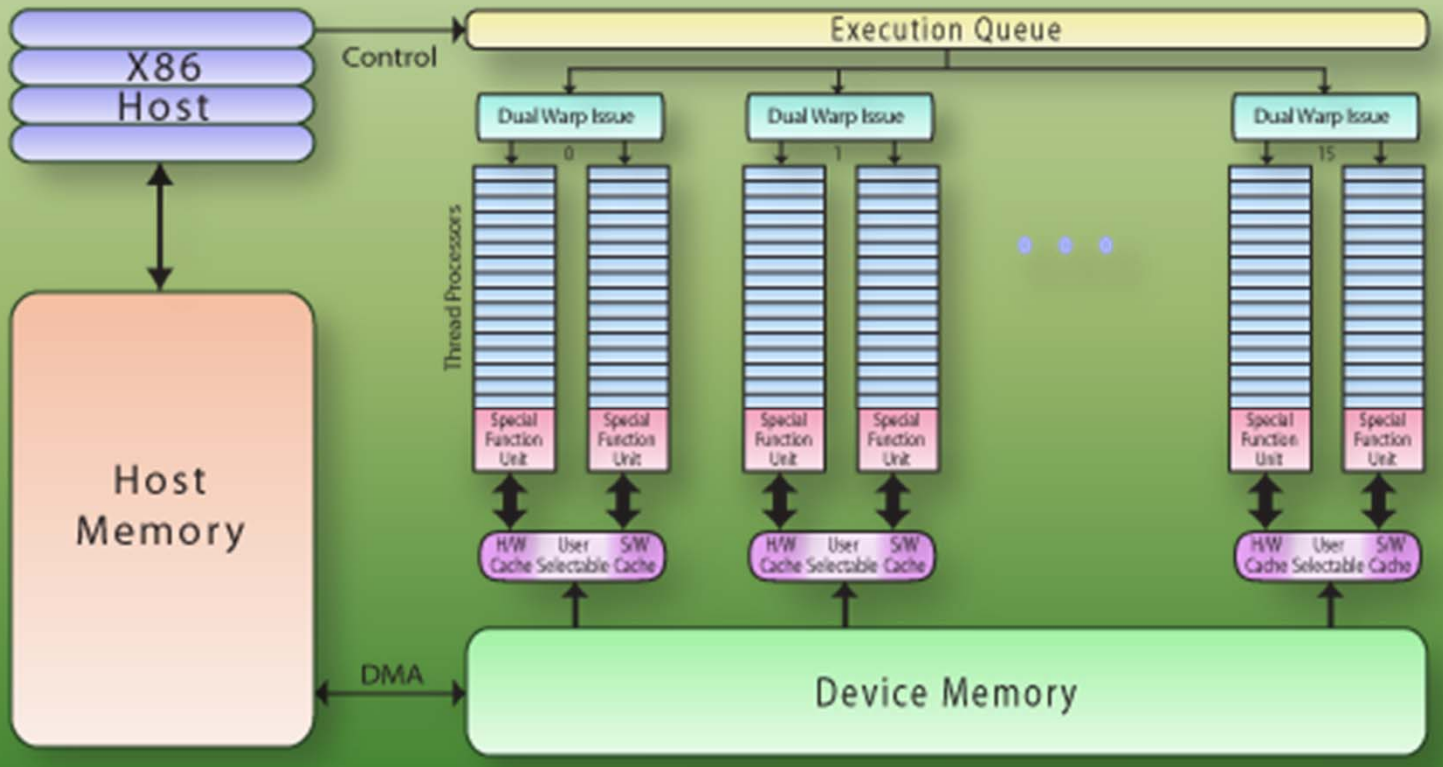


How Does It Work?



Different Architectures

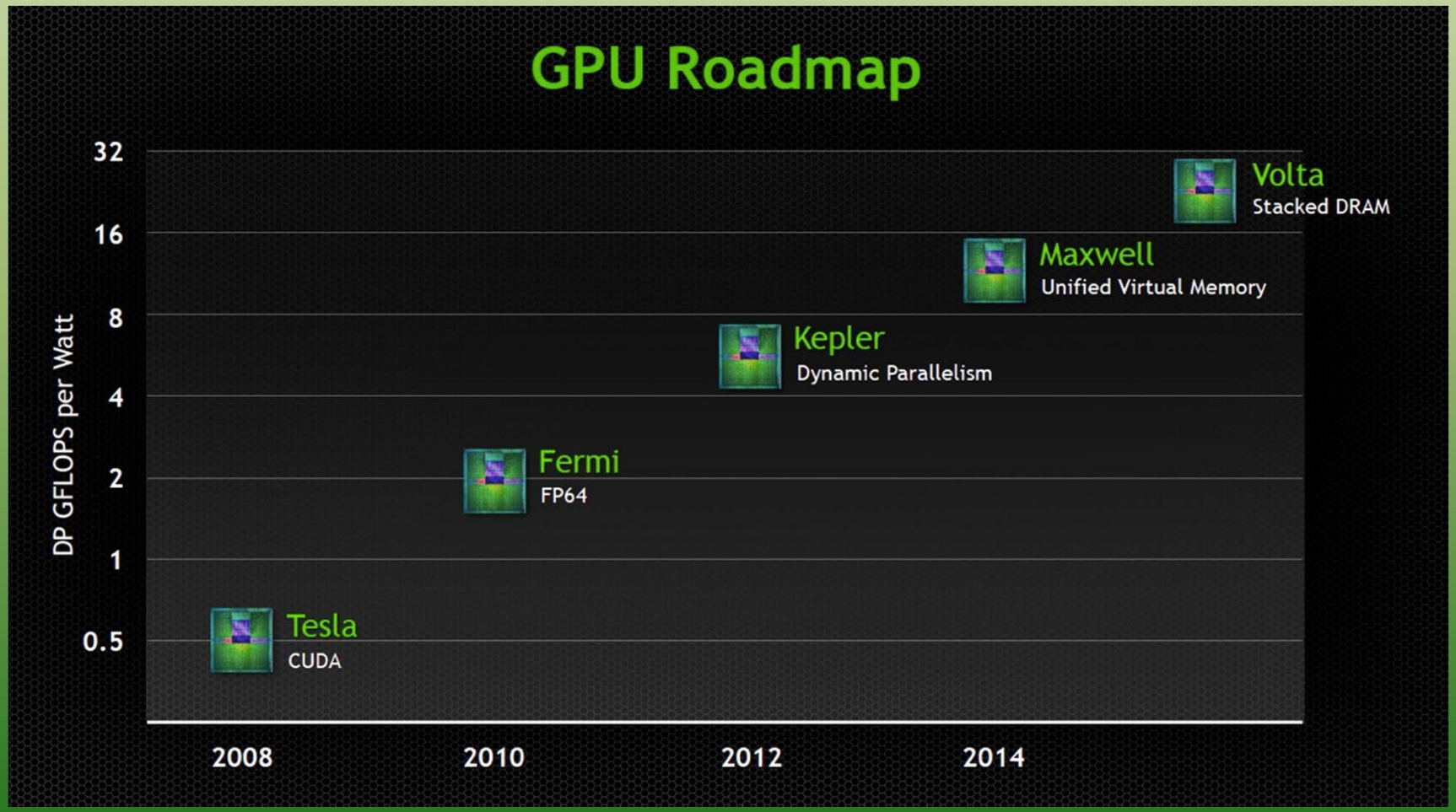
Fermi Architecture



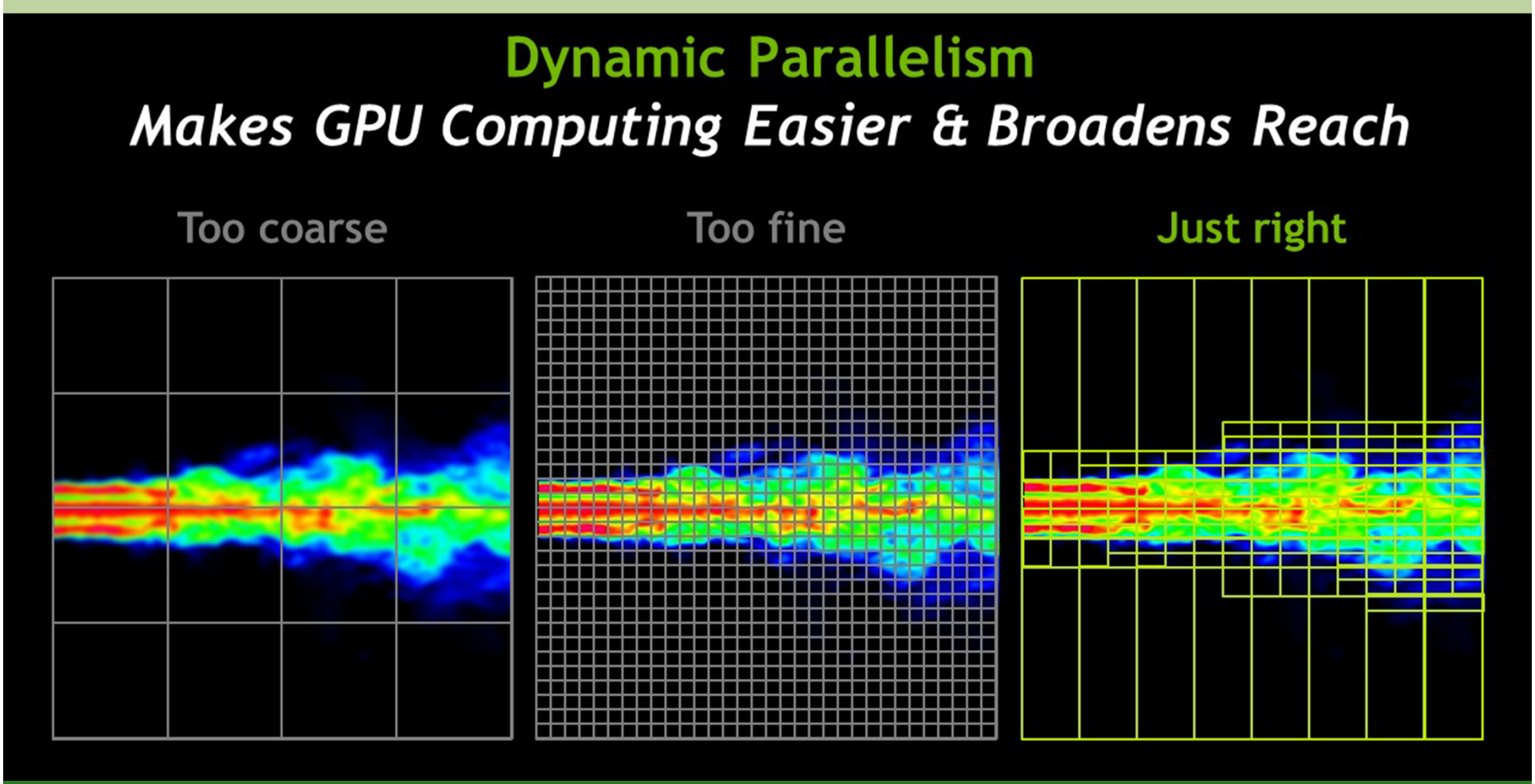
- More Multiprocessors
- Dual Warp Issue
- 4x Shared Memory (64 KB)

→ Tesla: 1 warp in 4 clock cycles
Fermi: 2 warps in 2 clock cycles

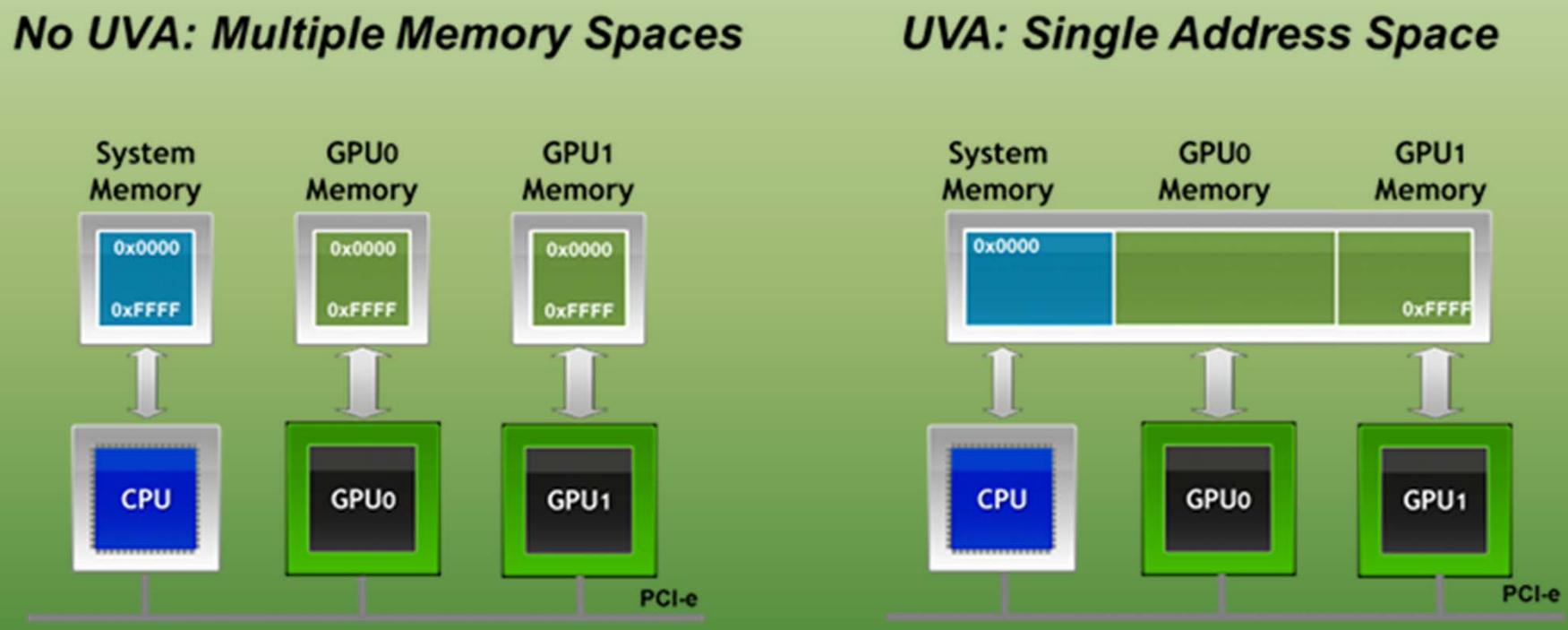
Evolution of Architectures



Dynamic Parallelism

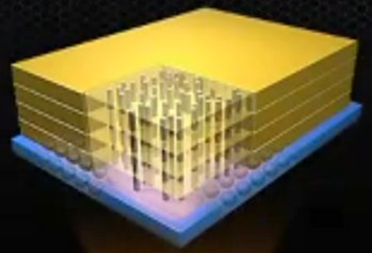


Unified Virtual Memory

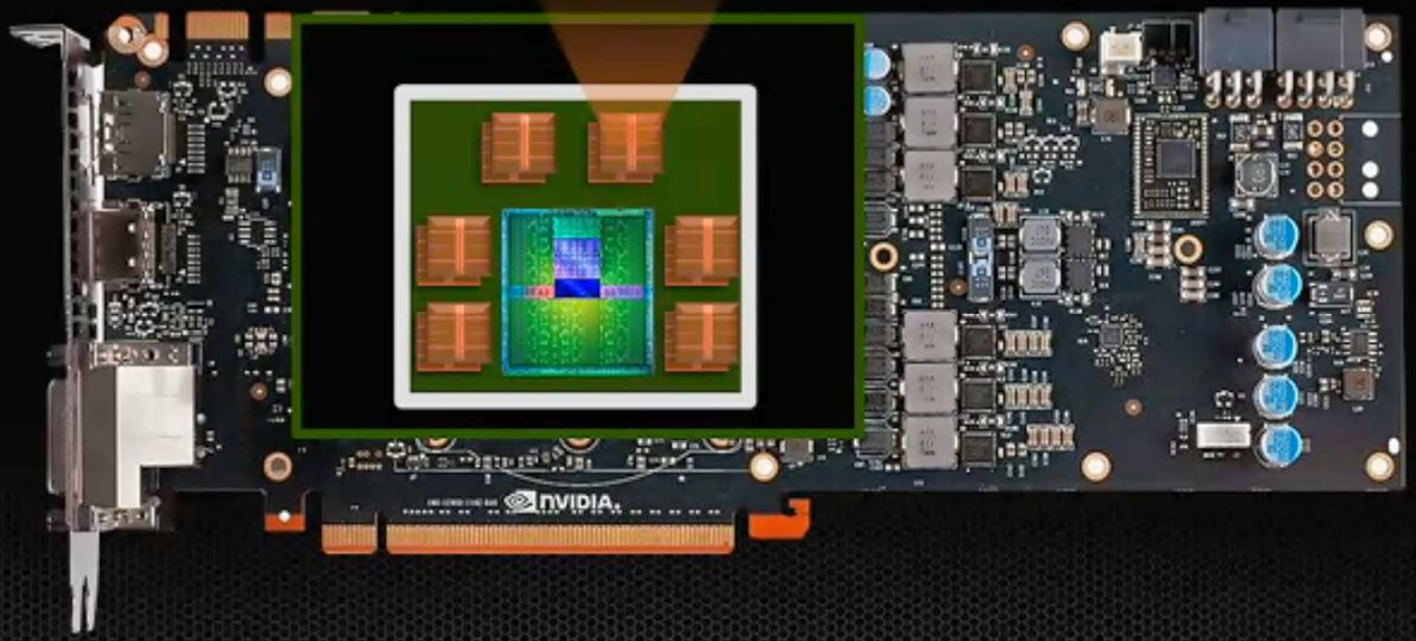


Volta with Stacked DRAM

Smaller Card Footprint



1 TB/s Memory Bandwidth



CUDA Programming

5 Parts of a CUDA Program:

1. Initialize Connection to GPU
2. Allocate Memory on GPU
3. Copy Data from RAM to GPU Memory
4. Execute Kernel
5. Copy Data from GPU Memory to RAM

Connecting to the GPU

Two Ways to Interface

1. Runtime API – Easier to Use, Less Control

```
#include <cuda_runtime_api.h>
cudaError success = cudaSetDevice(devID);
```

2. Driver API – More Code, Greater Control

```
#include <cuda.h>
CUdevice cuDevice;
CUcontext cuContext;
CUresult result = cuDeviceGet(&cuDevice, dev);
CUresult status = cuCtxCreate(&cuContext,
CU_CTX_SCHED_SPIN | CU_CTX_MAP_HOST, cuDevice);
```

**You can also write programs in PTX, an intermediate assembly language for NVIDIA GPUs. This is far more difficult.

Working with Memory

Allocating Memory

```
//Determine Memory Size (Ex:  10 doubles)
size_t mem_size = sizeof(double) * 10;
//Allocate Host Memory
double *h_data = (double*)malloc(mem_size);
//Initialize with your data
//Allocate Device Memory
double *d_data;
cudaError success =
cudaMalloc((void**)&d_data, mem_size);
//Copy Data from Host to Device
cudaError success = cudaMemcpy(d_data,
h_data, mem_size, cudaMemcpyHostToDevice);
```

Deallocating Memory

```
//Copy Result from Device to Host
cudaError success = cudaMemcpy(h_data,
d_data, mem_size, cudaMemcpyDeviceToHost);
//Free Host Memory
free(h_data);
cudaFree(d_data);
//Null Pointers
h_data = NULL;
d_data = NULL;
```

Make sure not to confuse host and device memory addresses!

Kernels

Kernels are functions written to run on the GPU:

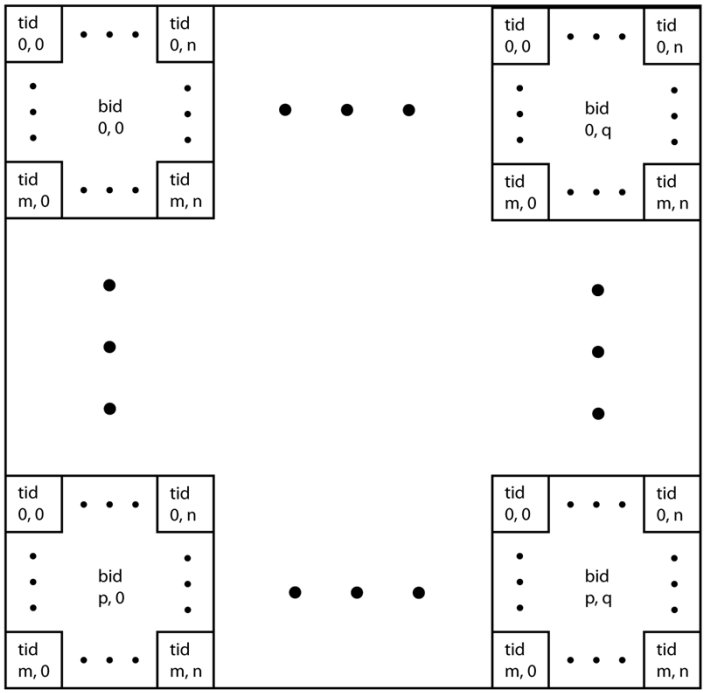
```
//This kernel adds the each value in d_data to its index
__global__ void myKernel(double *d_data)
{
        unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
        unsigned int j = blockDim.y * blockIdx.y + threadIdx.y;
        d_data[(i*width)+j] = d_data[(i*width)+j] + (i*width) + j;
}
```

Invocation on the host side executes the kernel:

```
//Kernel Parameters
dim3 threads(numthreads / blocksize, 1);
dim3 blocks(blocksize, 1);
//Execute Kernel
myKernel<<<blocks, threads>>>(d_data);
```

These determine the grid size

The Grid



A *kernel* launches a grid of thread blocks:

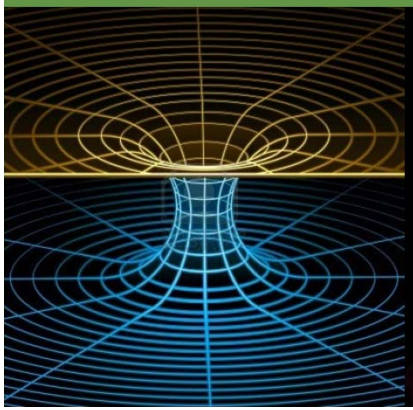
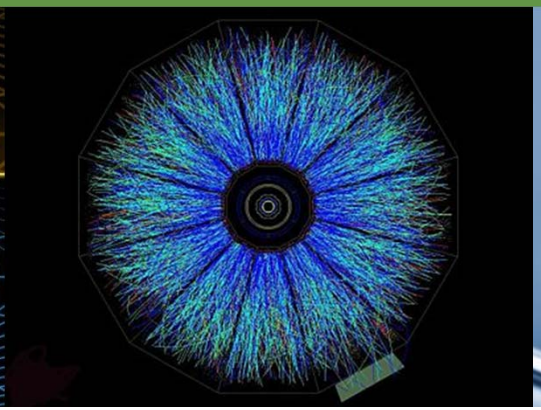
- Threads within a block cooperate
- Thread blocks can synchronize
- Each thread block is divided into 32 warps (multiprocessor scheduling units)
- All threads in a warp execute at same time

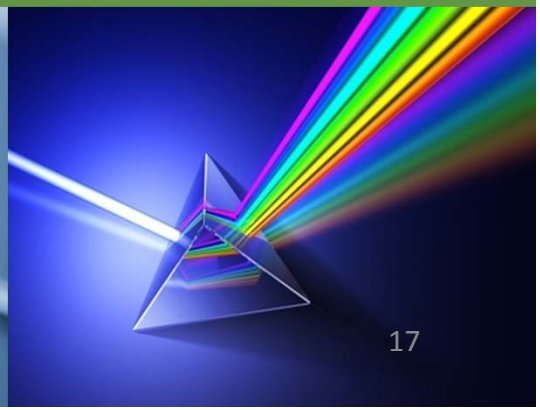
The structure of the grid reflects the nature of the GPU's hardware:

- GPUs were originally built to do 2D matrix algebra – pixels on a 2D monitor are updated all at the same time
- Take advantage of these features to better optimize your program

Application to Physics

- Nearly every physics problem can be related back to a *system of eigenvalue equations*
- This makes *matrix inversion* very important for computational physicists
- The algorithms become a bit more interesting when we are dealing with *sparse* matrices



The Eigenvalue Problem

$$Ax = \lambda b$$

Inertia Tensor (Cone):

$$I = \begin{bmatrix} \frac{3}{5}mh^2 + \frac{3}{20}mr^2 & 0 & 0 \\ 0 & \frac{3}{5}mh^2 + \frac{3}{20}mr^2 & 0 \\ 0 & 0 & \frac{3}{10}mr^2 \end{bmatrix} \qquad L = \begin{bmatrix} 0 \\ 0 \\ L_z \end{bmatrix}$$

$$I \cdot \omega = L$$

Now invert the inertial tensor to solve for the angular velocity eigenvalues!

Inversion on the CPU

There are many methods to decompose a matrix using a standard numerical algorithm. Many of these are optimized in downloadable libraries!

- Gauss-Jordan Elimination
- LU or QR Factorization
- Cholesky Decomposition
- Singular Value Decomposition
- Conjugate Gradient Method
- Biconjugate Gradient Method

Inversion on the GPU: The Sparse Conjugate Gradient Method

```
//Parameters Specific to CG Method
alpha = 1.0;
alpham1 = -1.0;
beta = 0.0;
r0 = 0.0;

//Perform the operation y = (d
cusparseScsrmv(cusparseHandl
CUSPARSE_OPERATION_NON_T
&alpha, descr, d_val, d_row, d_
//Perform the operation y = (d
cublasSaxpy(cublasHandle, N, &alpham1, d_Ax, 1, d_r, 1);
//Perform dot product operation
cublasSdot(cublasHandle, N, d_r, 1, d_r, 1, &r1);
```

Of course, it isn't always easy turning a textbook problem into a numerical algorithm…..

```
k = 1;
while  (r1 > tol*tol && k <= max_iter) {
      if (k > 1) {
            b = r1 / r0;
            cublasSscal(cublasHandle, N, &b, d_p, 1);
            cublasSaxpy(cublasHandle, N, &alpha, d_r, 1, d_p, 1);
      } else {
                                  sHandle, N, d_r, 1, d_p, 1);


                                  seHandle,
                                  _NON_TRANSPOSE, N, N, nz,
                                  d_row, d_col, d_p, &beta, d_Ax);
      cublasSdot(cublasHandle, N, d_p, 1, d_Ax, 1, &dot);
      a = r1 / dot;

      cublasSaxpy(cublasHandle, N, &a, d_p, 1, d_x, 1);
      na = -a;
      cublasSaxpy(cublasHandle, N, &na, d_Ax, 1, d_r, 1);

      r0 = r1;
      cublasSdot(cublasHandle, N, d_r, 1, d_r, 1, &r1);
      cudaThreadSynchronize();
      k++;
}
```

culaSgeqrf

Culapack Prefix	Data Type	Matrix Type	Computation Routine

External Resources

- NVIDIA Support (developer.nvidia.com)
- Download test matrices at Matrix Market (math.nist.gov/MatrixMarket)
- Popular Packages: CUBLAS, CULA, CUFFT, Thrust, CUSPARSE, CUSP
- Check out the Cuda Wiki (icmp.phys.rpi.edu/cudawiki) for more information, as well as instructions for how to use the GPU on your own computer for parallel programming!

Activity: Matrix Addition

Connect to CompPhys:

- ssh guest@compphys.phys.rpi.edu
- Password: 12345

Make Your Temporary Account:

- mkdir *rcsID*

Copy Files To Your Account:

- cp -r ./files/* ./*rcsID*/

Working with the Files:

- Only edit the file “addition.cu”
- To compile your code, use the command “make” while in the directory *rcsID.* Similarly, “make clean” clears old *cu_o files
- To run your program, use the command “./bin/Addition”

Try making a matrix multiplication kernel if you dare!