CUDA

Compute **U**nified **D**evice **A**rchitecture

Adam Simpson
Aug. 2011

Tesla X2090

- PCIe 2.0 interface
- 512 CUDA cores
- 665 Gigaflops peak DP performance
- 6 Gigabytes GDDR5 device memory
- 177 Gbytes/sec device memory bandwidth

Image provided by Nvidia

Fermi Architecture

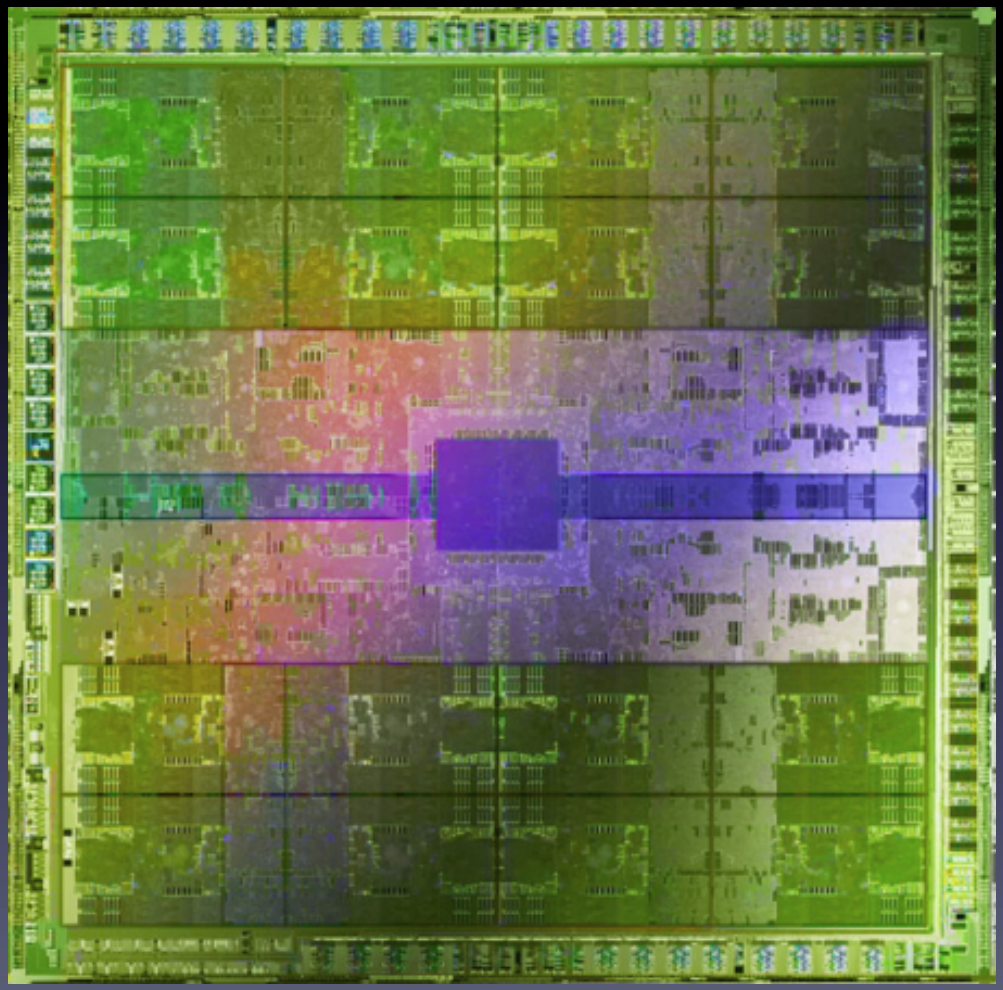
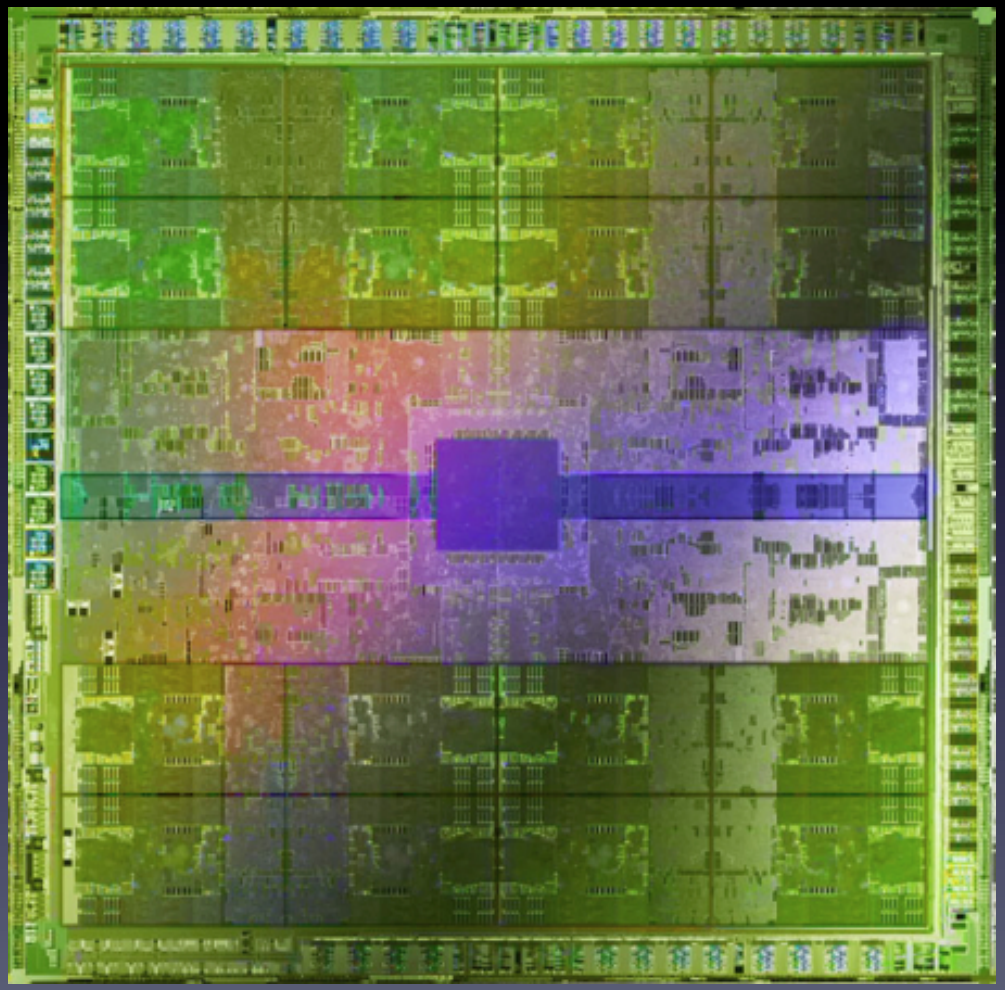


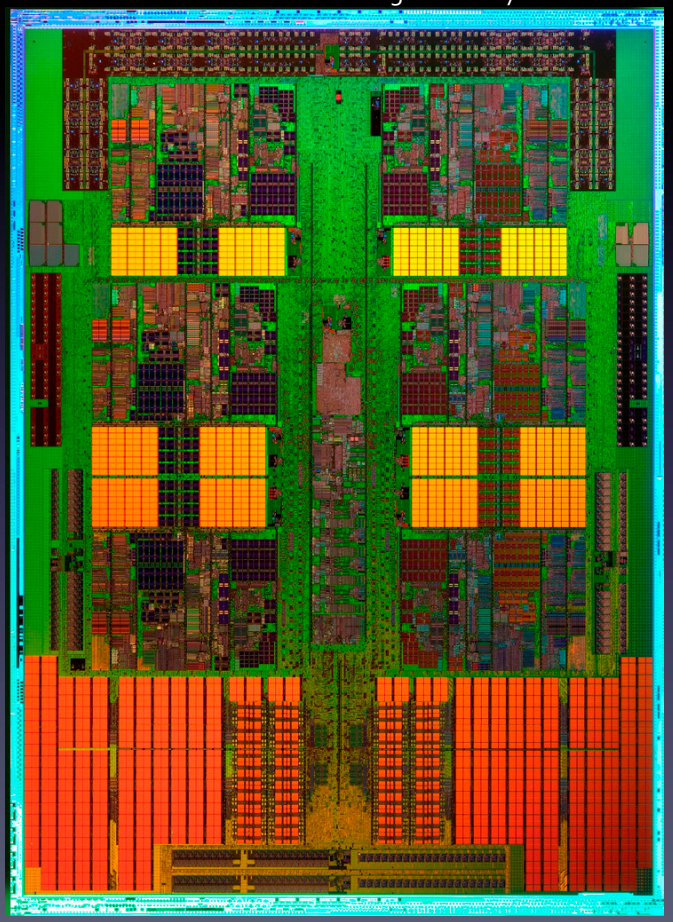
Image provided by Nvidia

Fermi vs 6 core AMD



530 mm²

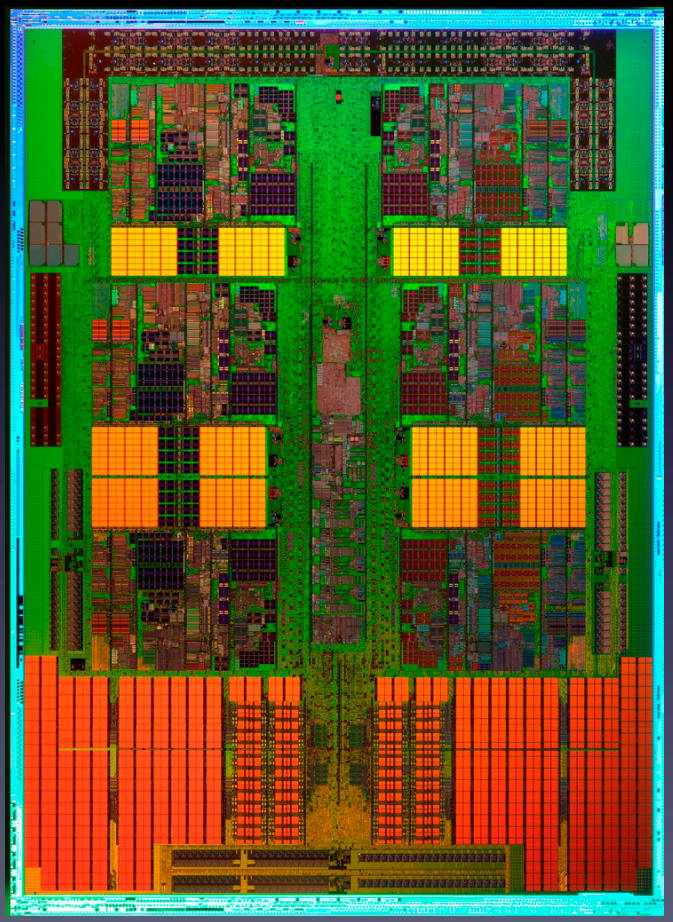
image courtesy of AMD



346 mm²

Fermi vs 6 core AMD

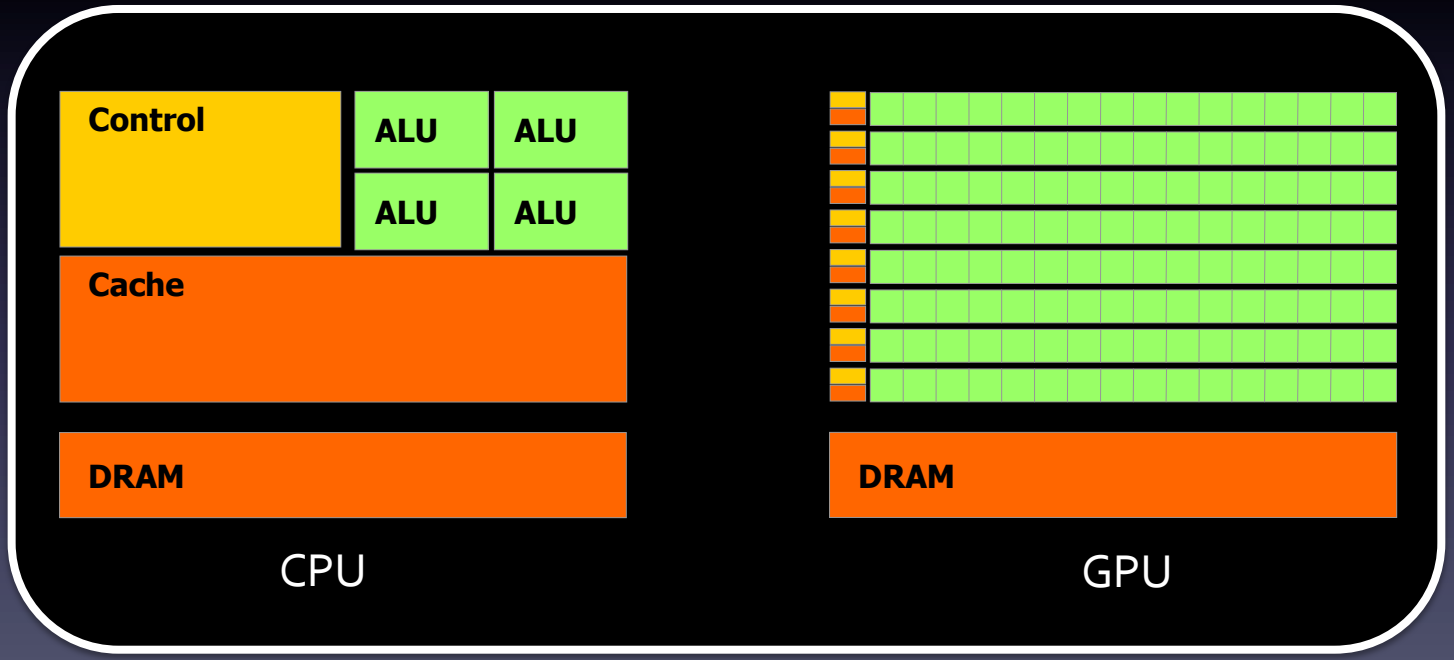
530 mm²



346 mm²

source: upload.wikimedia.org/wikipedia/commons/archive/a/a0/20090302225311%212006_Quarter_Proof.png

CPU vs GPU



source: cuda c programming manual

Fermi Overview

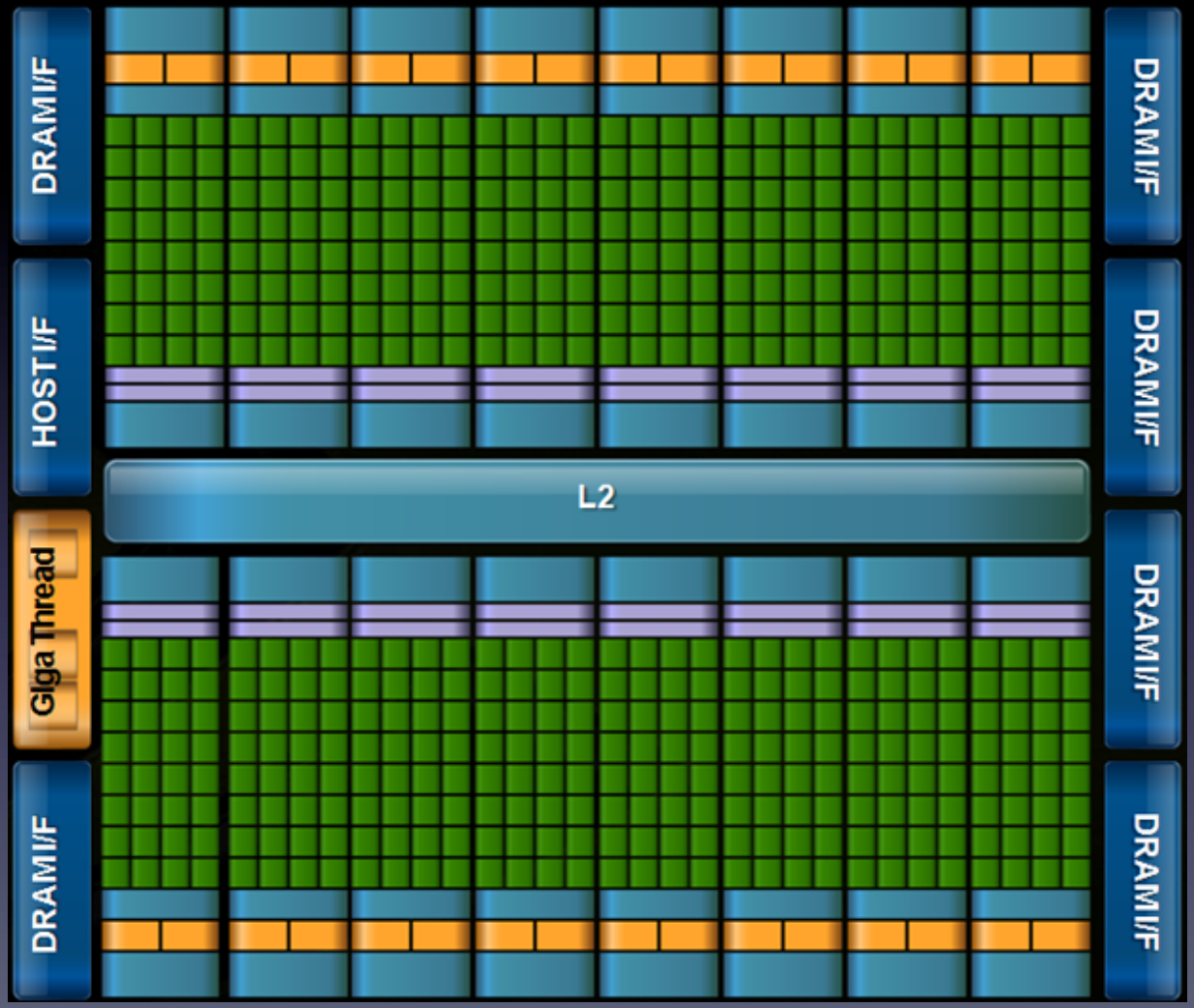


Image provided by nvidia

Fermi Overview

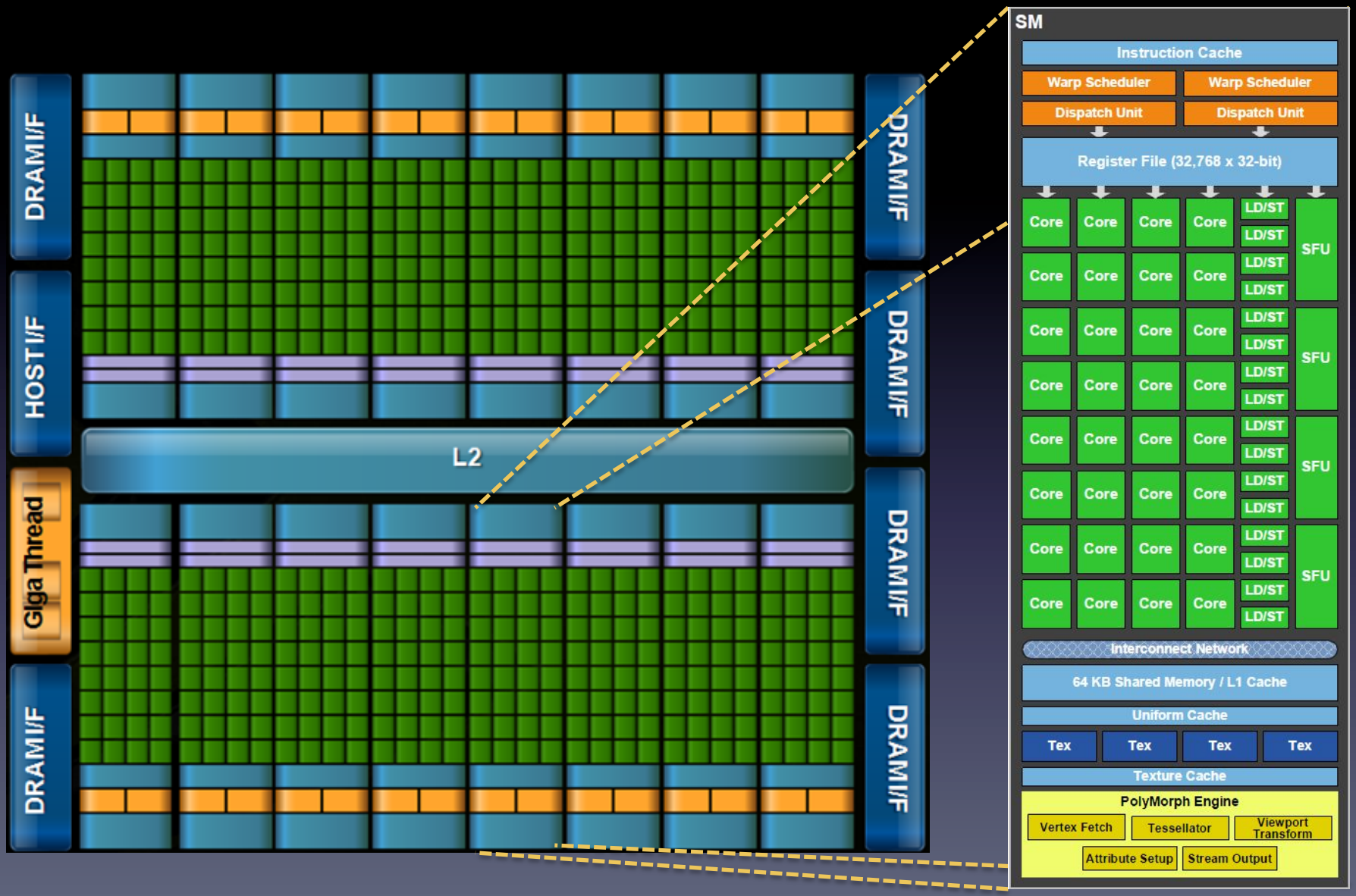
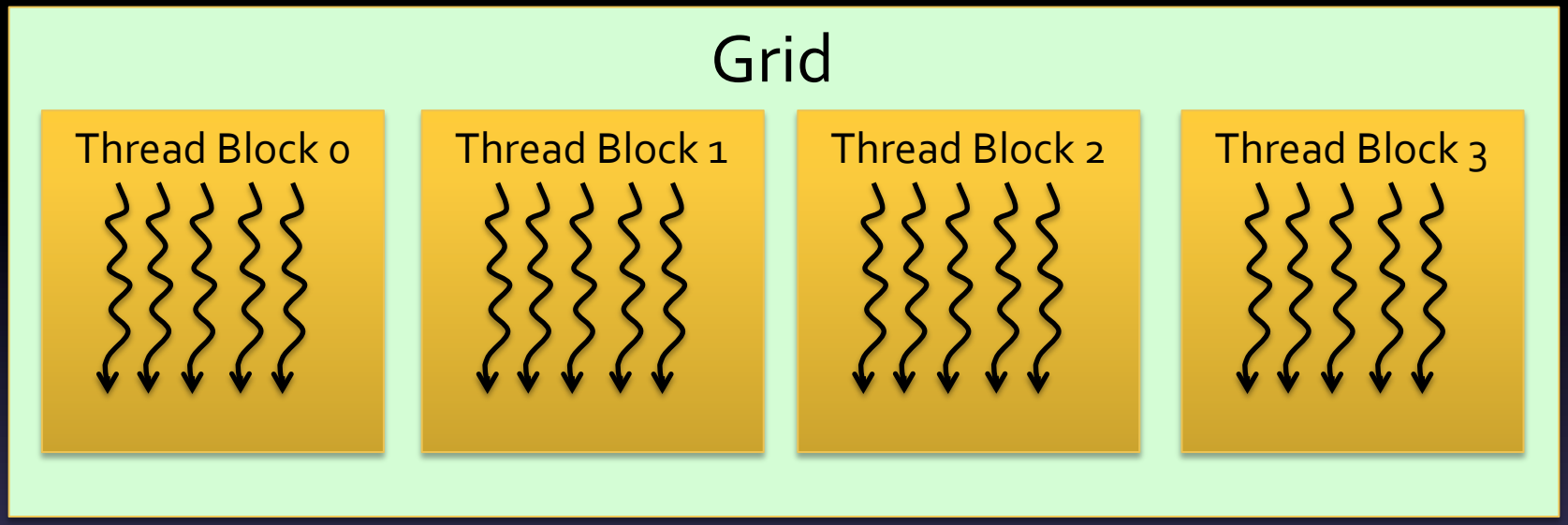
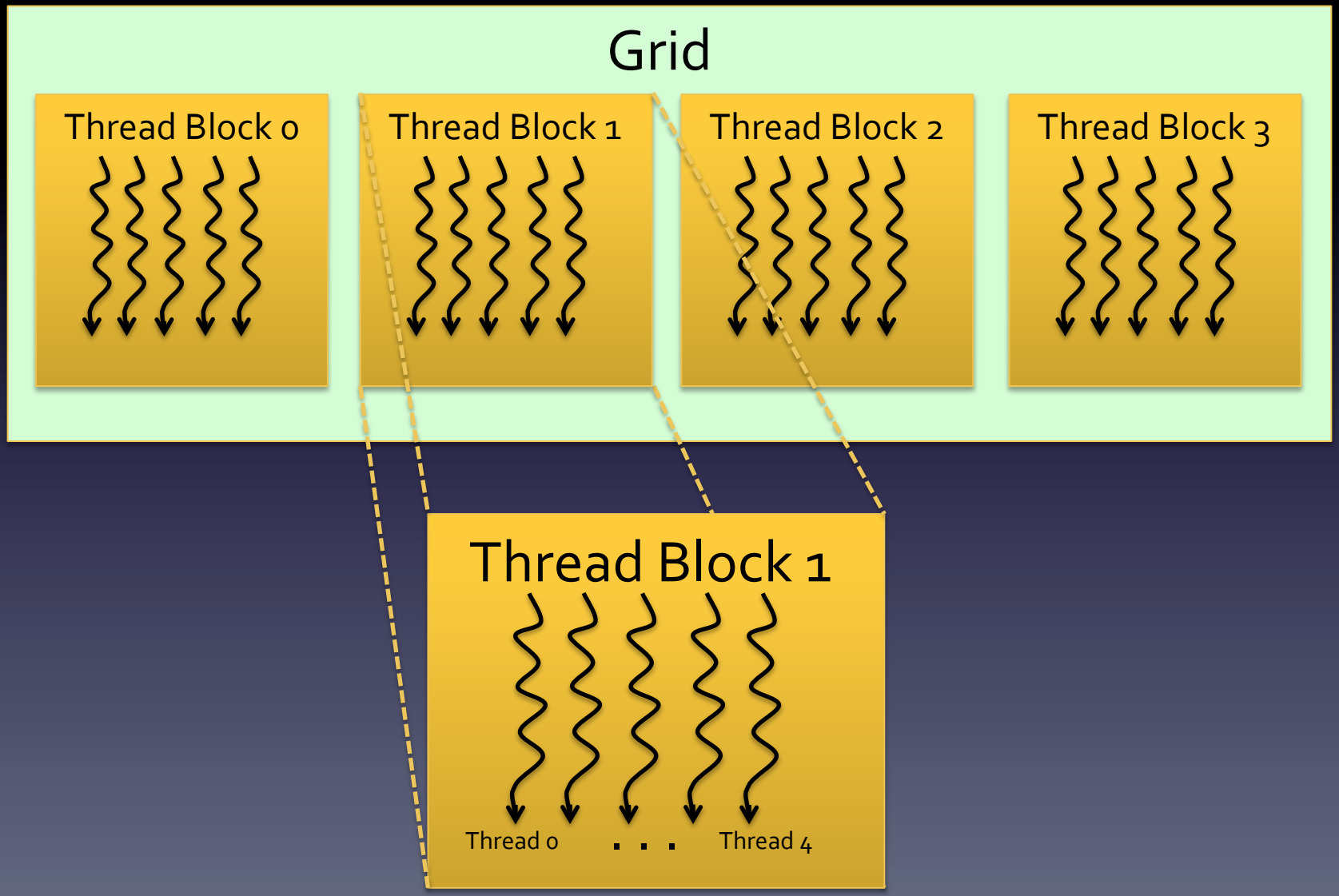


Image provided by nvidia

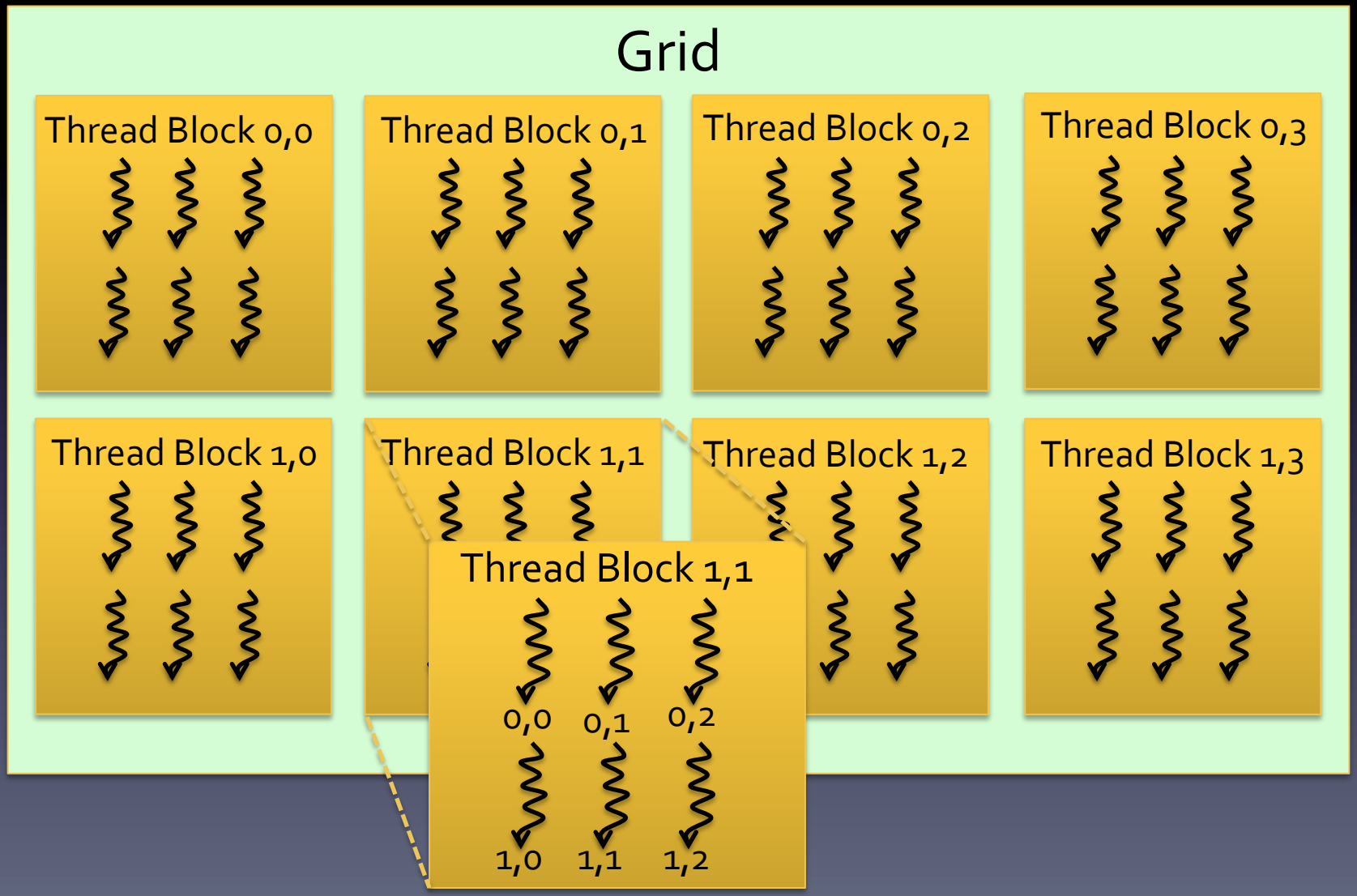
Thread Model



Thread Model



Thread Model



Memory Model

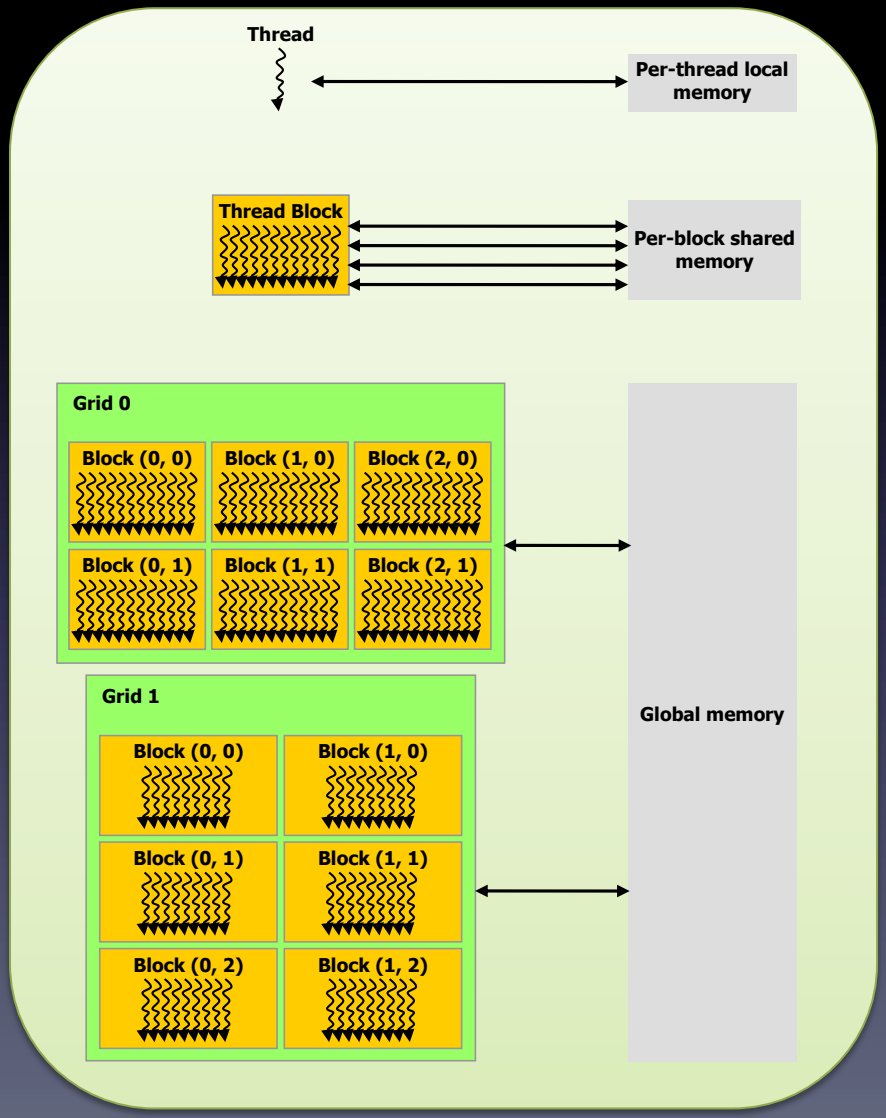


Image provided by nvidia

Memory Model

Memory Space	Latency	Size
Per thread register	1 cycle	128 KB per SM
Per block shared memory	2 cycles	16/48 KB per SM
Global memory with L1 and L2	600 cycles/Cached	6 GB : 768 KB L2 cache, 48/16 KB L1 per SM

Memory Model

Memory Space	Latency	Size
Per thread register	1 cycle	128 KB per SM
Per block shared memory	2 cycles	16/48 KB per SM
Global memory with L1 and L2	600 cycles/Cached	6 GB : 768 KB L2 cache, 48/16 KB L1 per SM
Texture cache	Cached	12KB cache per SM – shares 768 KB L2
Constant cache	Cached	64 KB : 8 KB cache per SM

CUDA C

- Minimal extension of C language
- Provided by NVIDIA for NVIDIA GPUs
- Explicitly control all data movement
- Basis of most other GPU tools

CUDA Kernel

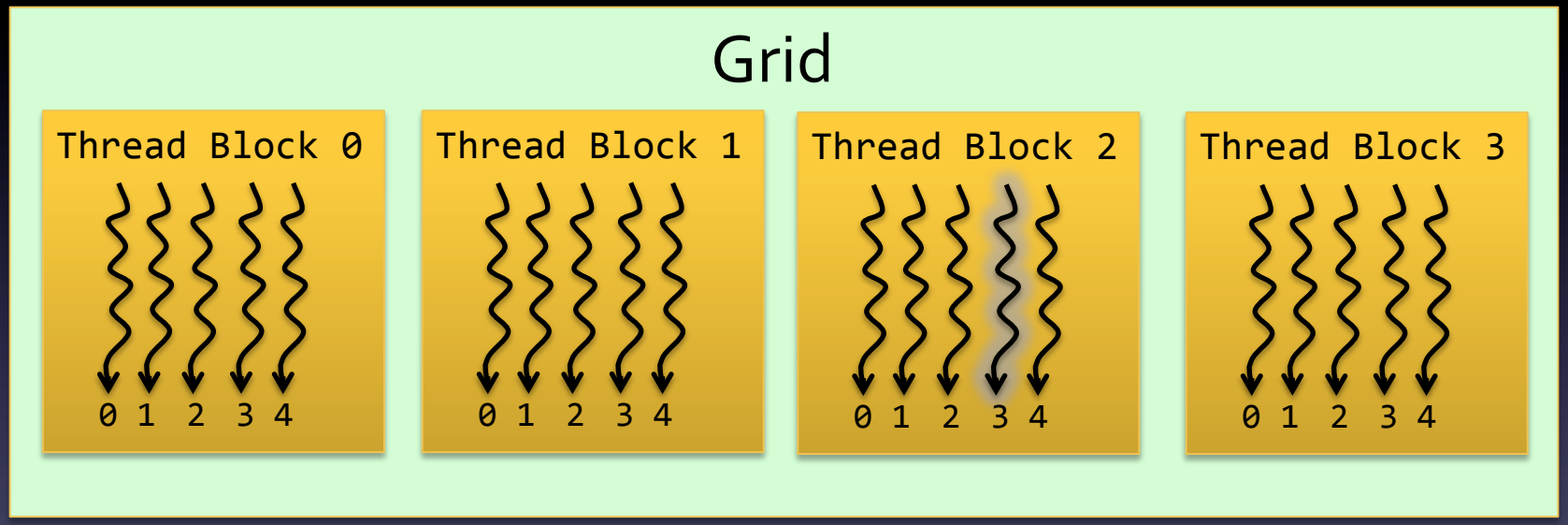
- C function executed N times in parallel by N CUDA thread
- Called from the host and executed on the GPU

```
//Kernel definition
__global__ void myKernel(float* A, int n)
{
    …
}

int main()
{
    //Kernel invocation
    VecAdd<<<blockSize,gridSize>>>(A,B,C);
}
```

CUDA Kernel

```
// Kernel definition
__global__ void square(float* A, int n)
{
    // Get our global thread ID
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    if(id < n)
        A[id] = A[id] * A[id];
}
```

Thread indexing




```
int id = blockIdx.x * blockDim.x + threadIdx.x;
```

blockIdx.x = 2
blockDim.x = 5
threadIdx.x = 3
id = 2*5 + 3 = 13

CUDA Memory

CPU

```
size_t bytes = n * sizeof(float);

// Allocate host memory
float *h_A;
h_A = (float *)malloc(bytes);

// Fill h_A with data and use
h_A[i] = 2.718; ...

// Release h_A
free(h_A);
```

GPU

```
size_t bytes = n * sizeof(float);

// Allocate device memory
float *d_A;
cudaMalloc((void**) &d_A , bytes);

// Copy from host memory to device
cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);

// Use on device
kernel<<<blockSize,gridSize>>>(d_A)

// Copy from device memory to host
cudaMemcpy(h_A, d_A, bytes, cudaMemcpyDeviceToHost);

// Release memory
cudaFree(d_A);
```

Setup and Launch Kernel

```
// Number of threads in each thread block
dim3 blockSize(256,1,1);

// Number of thread blocks in grid
int blocks = (int)ceil((float)n/blockSize.x);
dim3 gridSize(blocks,1,1);

// Execute the kernel
square<<<gridSize, blockSize>>>(d_A, n);
```

VecAdd.c

```c
#include <stdlib.h>
#include <math.h>

int main( int argc, char *argv[] )
{
    // Length of array
    int n = 100000;

    // Input arrays
    float *a;
    float *b;
    // Output array
    float *c;

    // Size, in bytes, of each array
    size_t bytes = n*sizeof(float);

    // Allocate memory for each array
    a = (float*)malloc(bytes);
    b = (float*)malloc(bytes);
    c = (float*)malloc(bytes);

    // Initialize content of input array
    int i;
    for(i=0; i<n; i++) {
        a[i] = sinf(i)*sinf(i);
        b[i] = cosf(i)*cosf(i);
    }
```

```c
    // Sum component wise and save result
    for(i=0; i<n; i++){
        c[i] = a[i] + b[i];
    }

    // Release memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

VecAdd.cu

```
#include <stdlib.h>
#include <math.h>

// Write CUDA kernel

int main( int argc, char *argv[] )
{
    // Length of array
    int n = 100000;

    // Declare and allocate device arrays

    // Input arrays
    float *a;
    float *b;
    // Output array
    float *c;

    // Size, in bytes, of each array
    size_t bytes = n*sizeof(float);

    // Allocate memory for each array
    a = (float*)malloc(bytes);
    b = (float*)malloc(bytes);
    c = (float*)malloc(bytes);

    // Initialize content of input array
    int i;
    for(i=0; i<n; i++) {
        a[i] = sinf(i)*sinf(i);
        b[i] = cosf(i)*cosf(i);
    }
```

~~// Sum component wise~~
~~for(i=0; i<n; i++)~~
~~c[i] = a[i] + b[i];~~

```
    // Copy input arrays to device

    // Setup and launch kernel

    // Copy output array to host

    // Test results for correctness

    // release device memory

    // Release memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

NVCC

- Used to compile mixed C/CUDA .cu files
- Provided with CUDA toolkit
- C portion of code compiled with GCC
- Takes flags similar to GCC

Compiling and Running

```
$ ssh user@lens.ccs.ornl.gov
$ module load cuda
$ qsub –A### -I –V –lwalltime=00:30:00,nodes=1:ppn=1
$ nvcc VecAdd.cu -o add.out
$ ./add.out
```

Additional Info

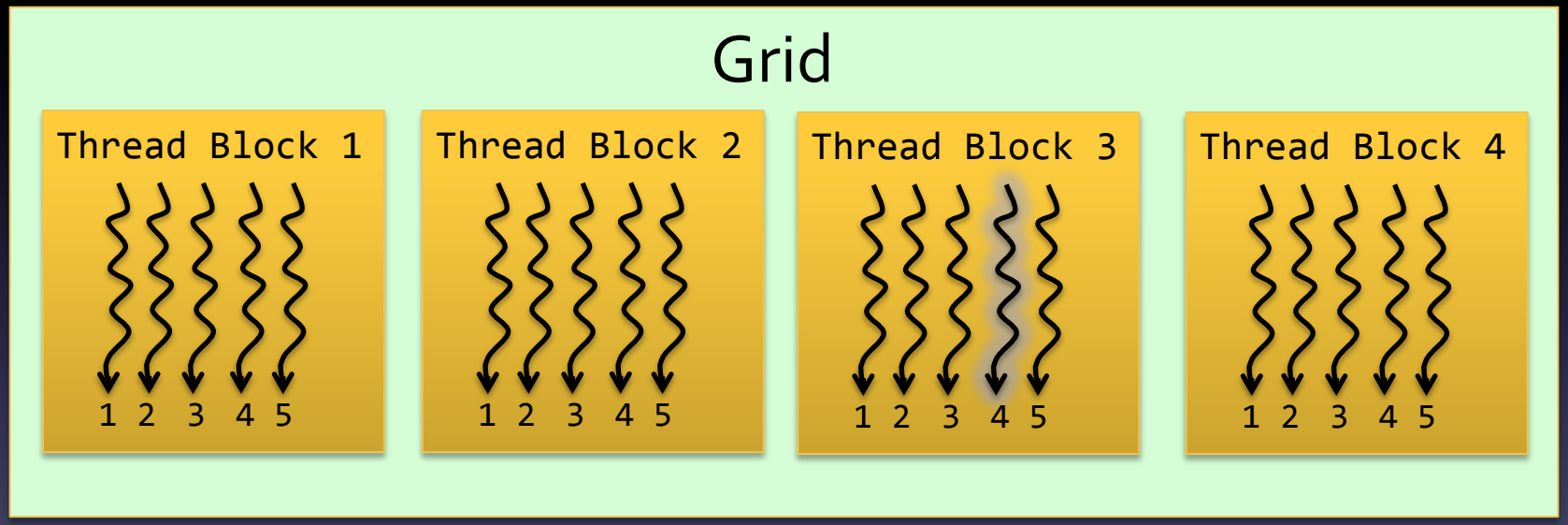
https://developer.nvidia.com/category/zone/cuda-zone

- **CUDA C Programming guide**
- Fermi tuning and CUDA C best practices
- CUDA SDK

CUDA Fortran

- Provided by PGI
- License required
- Mirrors *most* features of CUDA C
- Presented as addition to Fortran as opposed to module making some operations implicit

Thread indexing




```
id =  (blockidx%x-1)*(blockdim%x) + threadidx%x
```

blockidx%x = 3
blockdim%x = 5
threadIdx.x = 4
id = (3-1)*5 + 4 = 14

CUDA Fortran Kernels

```
! Kernel definition
Attribute(global) subroutine square(n, A)
    integer, value :: n
    real, device :: A(n)
    integer :: id
    id =  (blockidx%x-1)*(blockdim%x) + threadidx%x

    if(id <= n) then
        A(i) = A(i) * A(i)
    endif
end subroutine square

program main
    ! Kernel invocation
    call square<<<blockSize, gridSize>>>(n, A);
end program main
```

CUDA Fortran Memory

```fortran
! Allocate and initialize host memory
real, allocatable :: h_A
allocate(h_A(n))

! Allocate device memory
real, allocatable, device :: d_A
allocate(d_A(n))

! Copy from host memory to device
d_A = h_A

! Copy from device memory to host
h_A = d_A

! Release memory
deallocate(h_A)
deallocate(d_A)
```

Setup and Launch Kernel

```
type(dim3) :: blockSize, gridSize

! Number of threads in each thread block
blockSize = dim3(256,1,1);

! Number of thread blocks in grid
gridSize = dim3(ceiling(real(n)/real(blockSize%x)) ,1,1)

! Execute the kernel
call square<<<gridSize, blockSize>>>(n,  d_A);
```

VecAdd.cuf

```fortran
module kernel
    contains
    ! CUDA Fortran kernel
end module kernel

program main
    use cudafor
    use kernel

    ! Size of vectors
    integer :: n = 100000

    ! Declare and allocate device arrays

    ! Input vectors
    real,dimension(:),allocatable :: a
    real,dimension(:),allocatable :: b
    ! Output vector
    real,dimension(:),allocatable :: c

    integer :: i
    real :: sum

    ! Allocate memory for each vector
    allocate(a(n))
    allocate(b(n))
    allocate(c(n))
```

```fortran
! Initialize content of input vectors
do i=1,n
    a(i) = sin(i*1.0)*sin(i*1.0)
    b(i) = cos(i*1.0)*cos(i*1.0)
enddo

! Copy input arrays to device

! Setup and launch kernel

! Copy output array to host

! Test results for correctness

! Release device memory

! Release memory
deallocate(a)
deallocate(b)
deallocate(c)

end program
```

Compiling and Running

```
$ ssh user@lens.ccs.ornl.gov
$ qsub –A### -I –V –lwalltime=00:30:00,nodes=1:ppn=1
$ pgfortran VecAdd.cuf -o add.out
$ ./add.out
```

Additional Resources

- http://www.pgroup.com/resources/cudafortran.htm
- CUDA Fortran Programming Guide

Homework

- Fill in each comment of VecAdd.cu/VecAdd.cuf
- Compile and run on lens
- We will review the problem tomorrow