

# Introduction to CUDA-C and CUDA-FORTRAN programming

Raj Shukla

RCC  
University of Chicago

July 23, 2020

# GPU really?

*Arithmetic is cheap.*

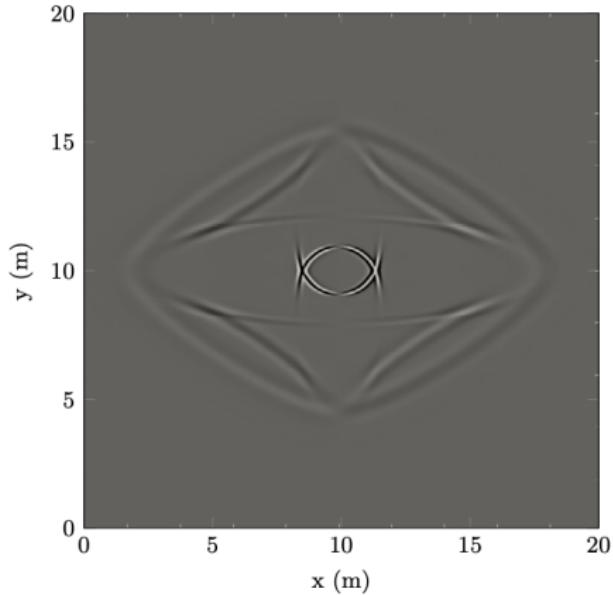
*Latency is physics.*

*Bandwidth is money.*

-Mark Hoemmen\*

\* Sandia National Laboratories

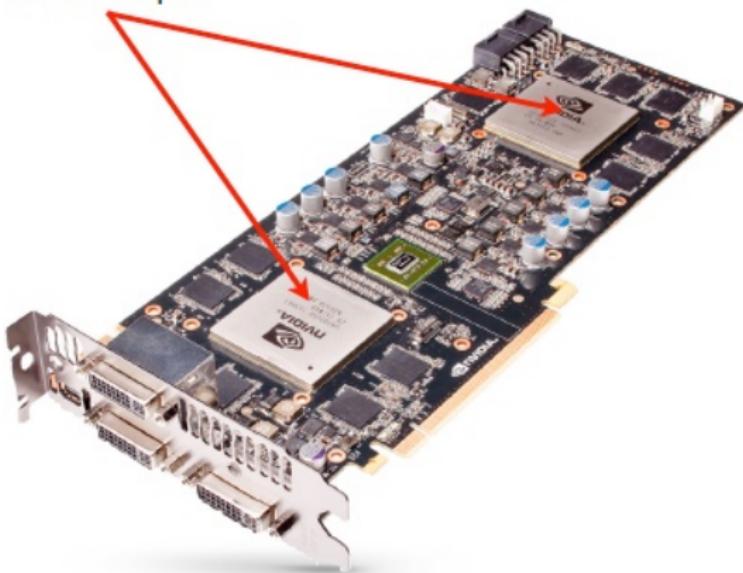




(a) Epoxy-glass,  $b_x$  with  $\eta = 0$

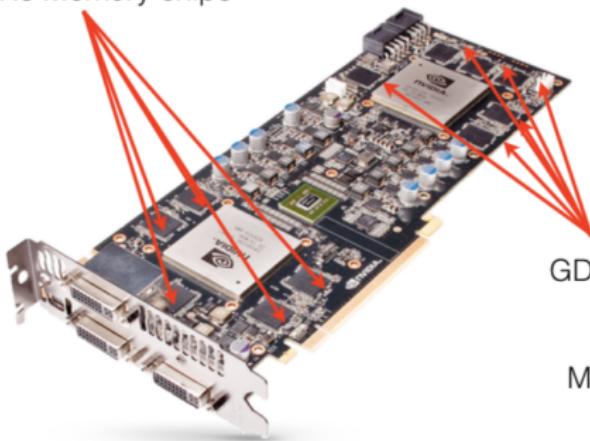
# Example GPU Card: GTX590 GPU

Two Fermi (GF110)  
GPU chips



# Example GPU Card: GTX590 GPU

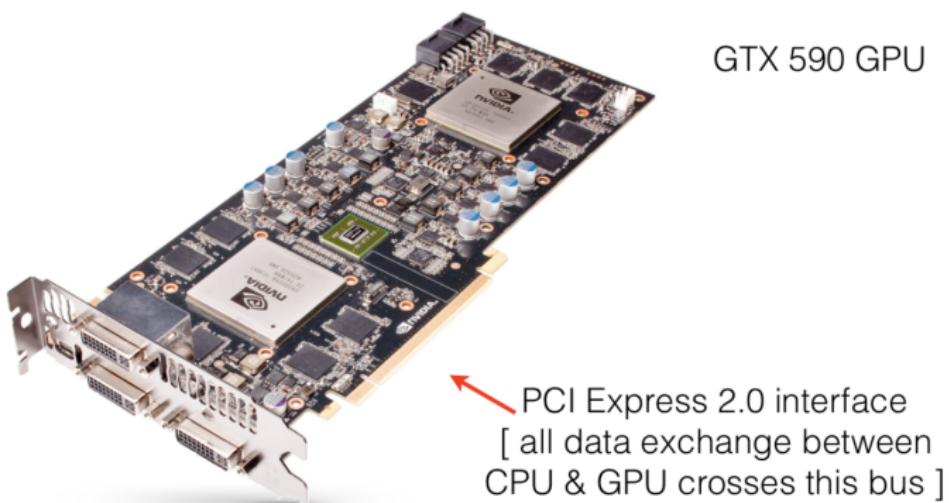
GDDR5 Memory chips



GDDR5 Memory chips:  
1536MB

Memory bus: 384 bit

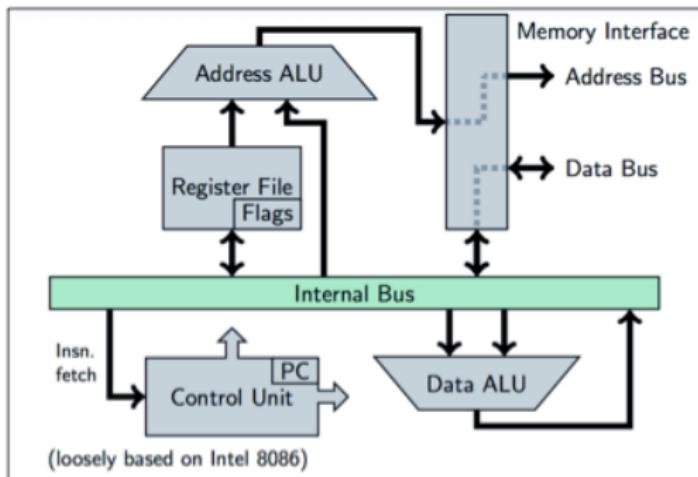
# Example GPU Card: GTX590 GPU



# CPU: A simplified architecture

## Design goals of CPUs

- Make a single thread very fast
  - 1 Reduce latency through large caches.
  - 2 predict, speculate.



# ALU

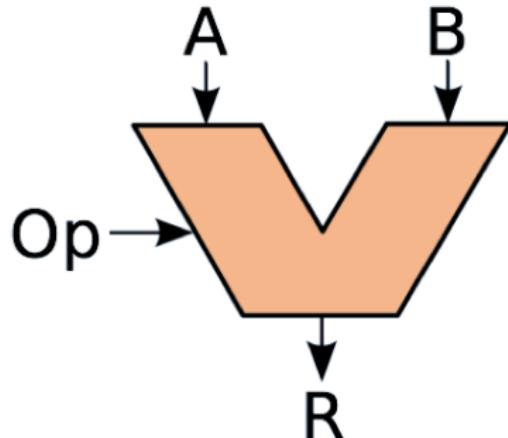
One or two operands A, B?

Operation selector (Op):?

- (Integer) Addition, Subtraction.
- (Integer) Multiplication, Division.
- (Logical) And, Or, Not.
- (Bitwise)

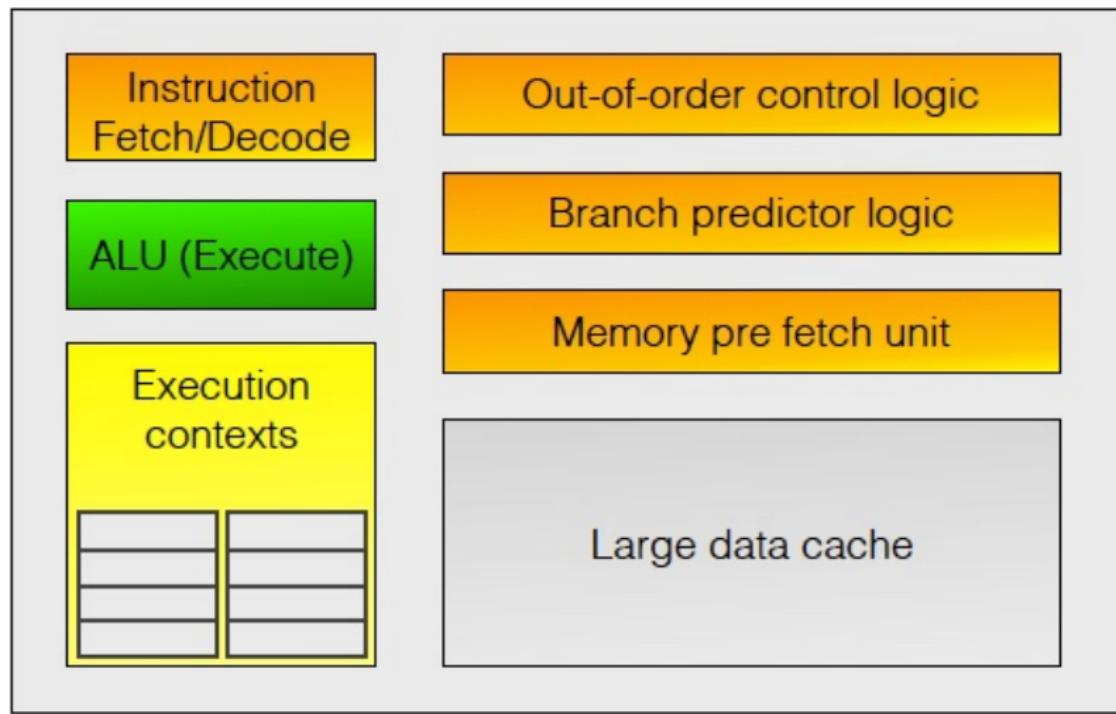
ALUs

Floating Point Unit (FPU)  
Address ALU.



# CPU: Abstract Architecture

*Modern “CPU-Style” core design emphasizes individual thread performance.*



# Latency vs Throughput

## Going from point A to B: Distance 4500 km

Car: 2 People

Bus: 40 People

Speed: 200 km/h

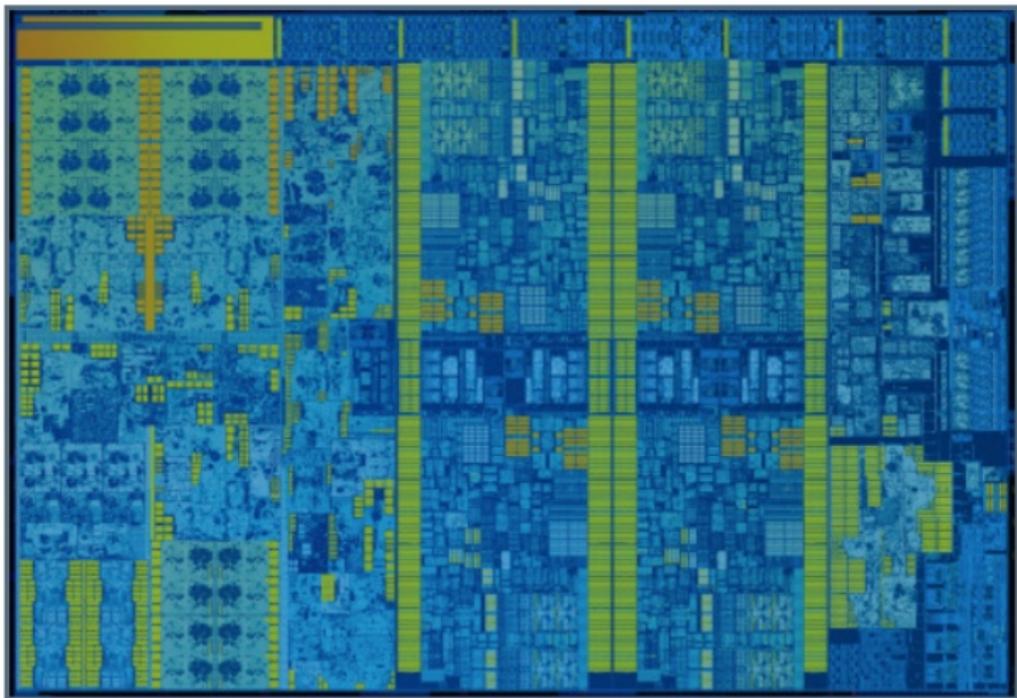
Speed: 50 km /h

Latency= 22.5 Hours

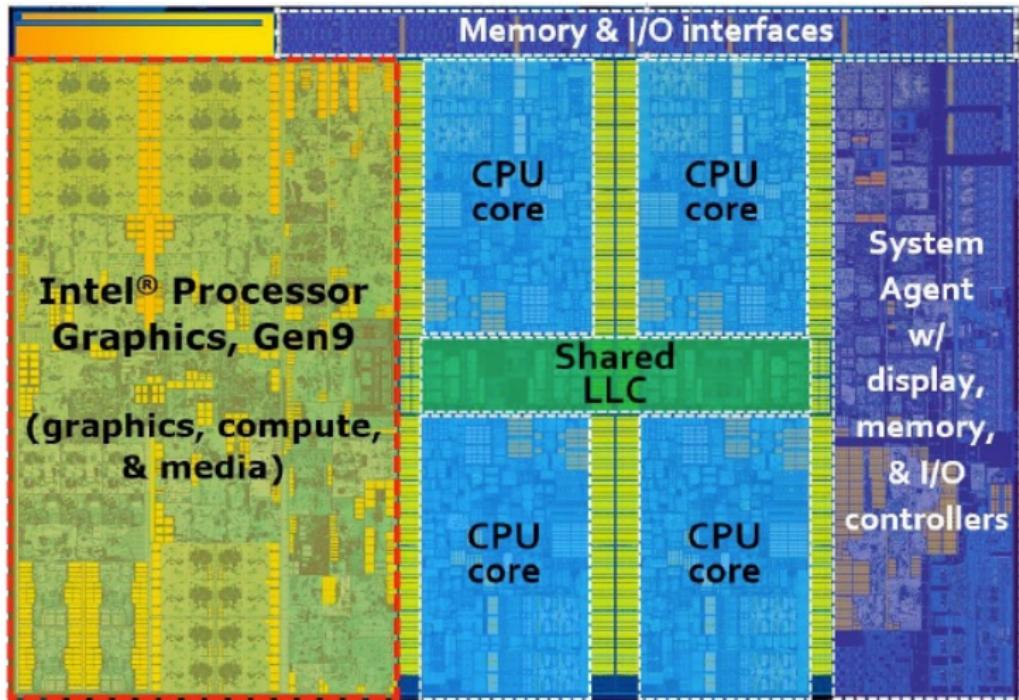
Latency: 90 Hours

Throughput = 0.089 People/Hour      Throughput = 0.45 People/Hour

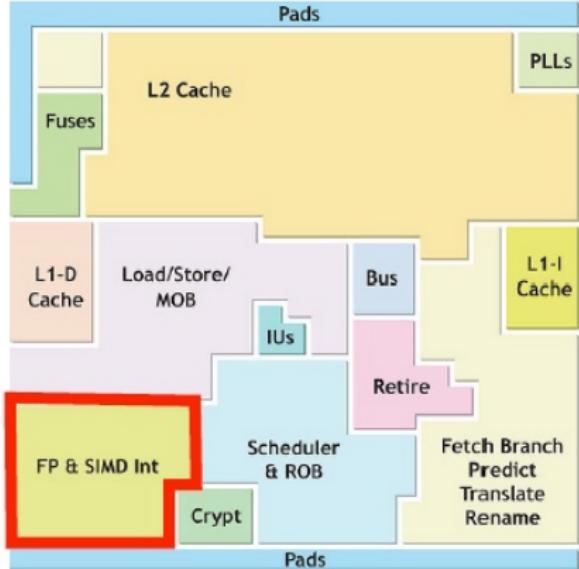
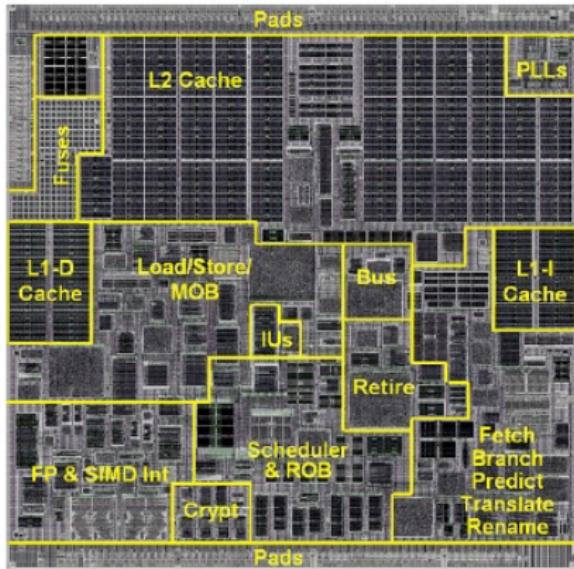
# CPU:Intel Skylake Architecture



# CPU:Intel Skylake Architecture

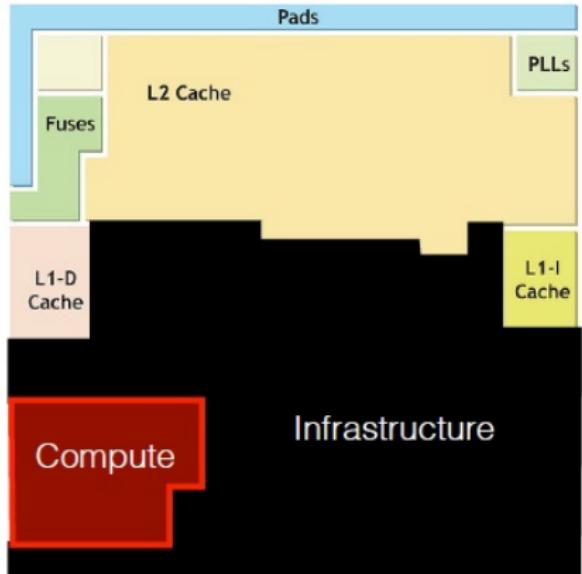
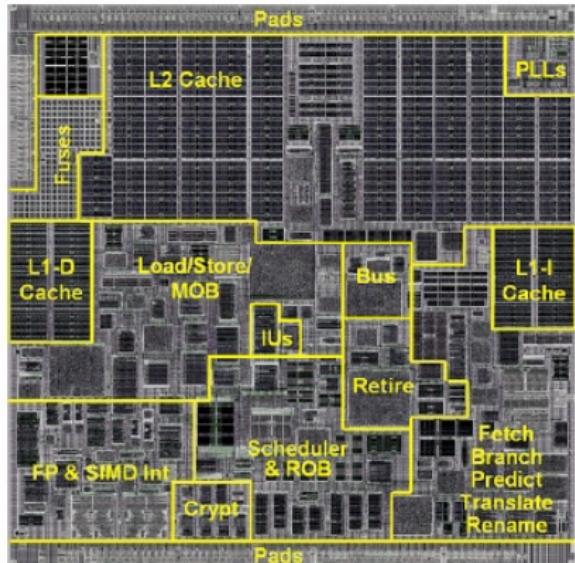


# CPU:Floor Plan



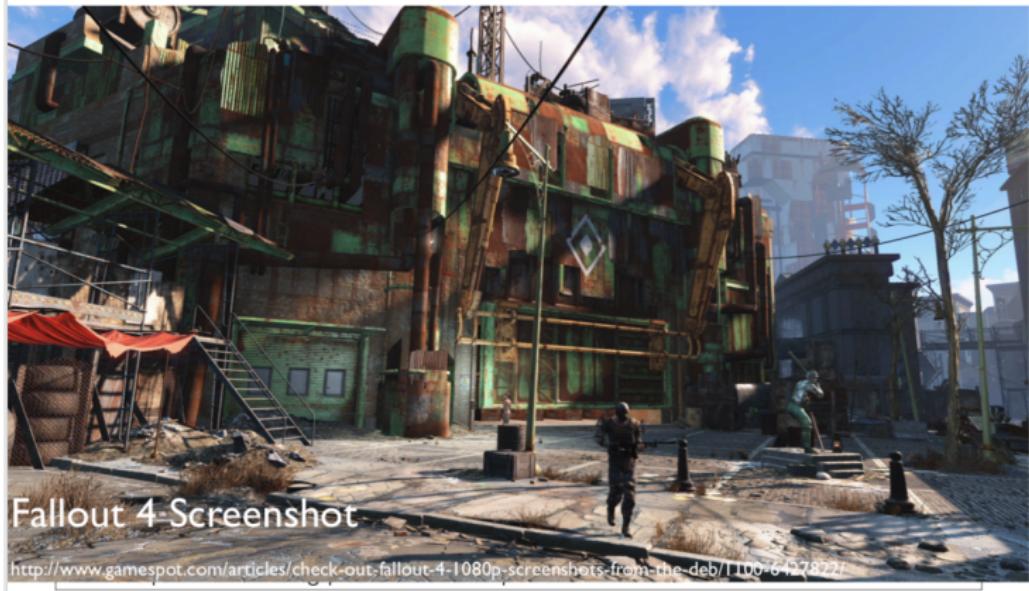
Die floorplan: VIA Isaiah (2008). 65 nm, 4 SP ops at a time, 1 MiB L2.

# CPU:Floor Plan



# GPUs

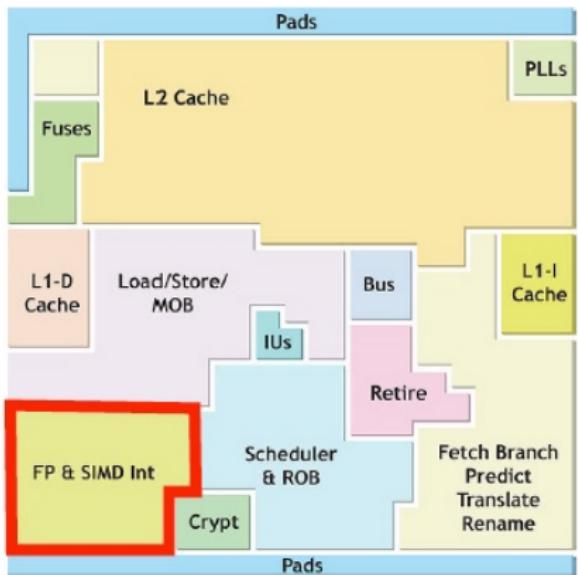
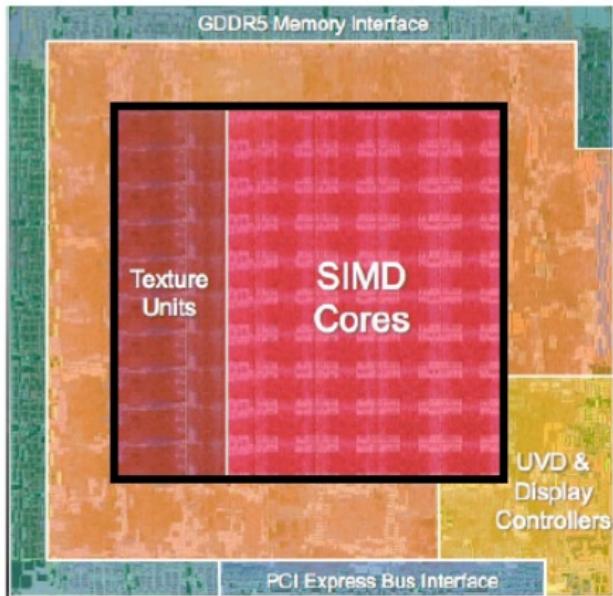
The main purpose of graphics processing units is to project textured polygons onto the screen.



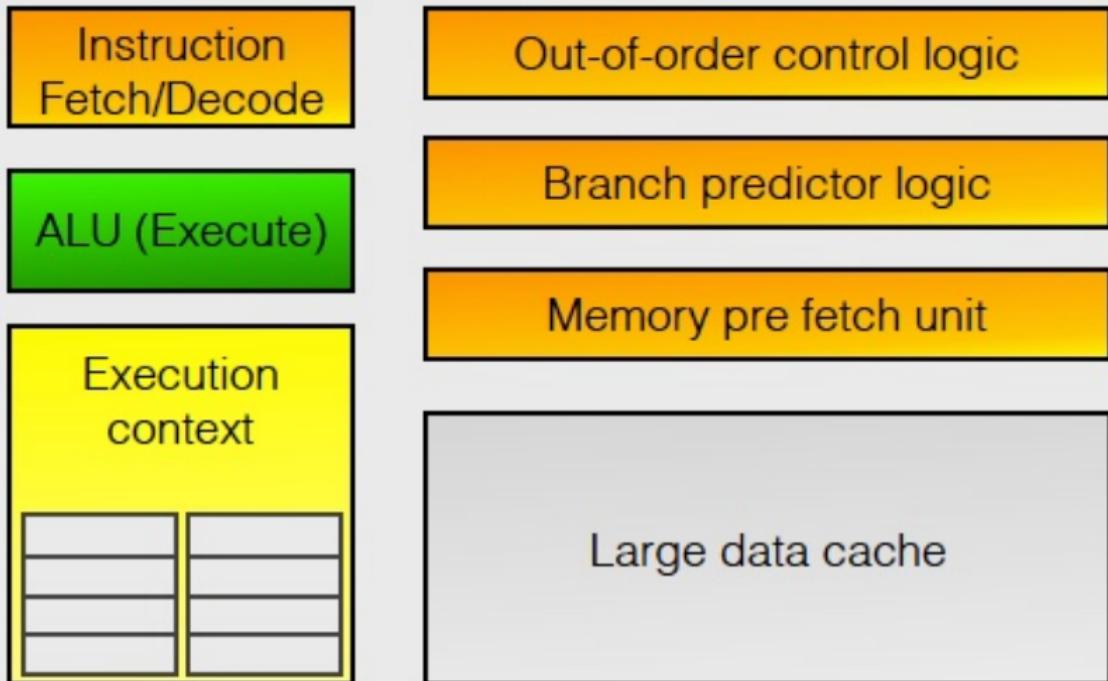
# Design goals of GPUs

- Throughput matters, single thread does not
- Hide memory latency through parallelism
- Let programmer deal with "raw storage of memory hierarchy"
- Avoid high frequency clock speed

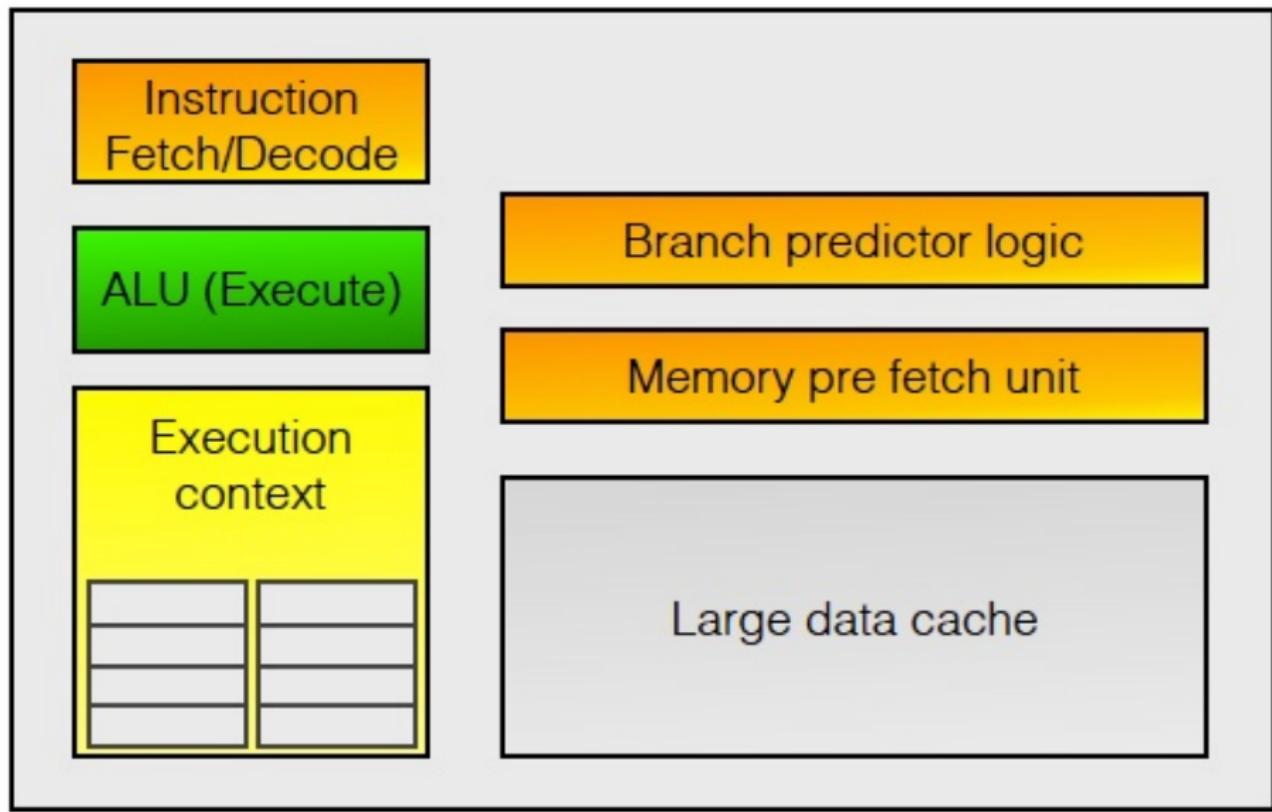
# GPU vs CPU: Floor Plan



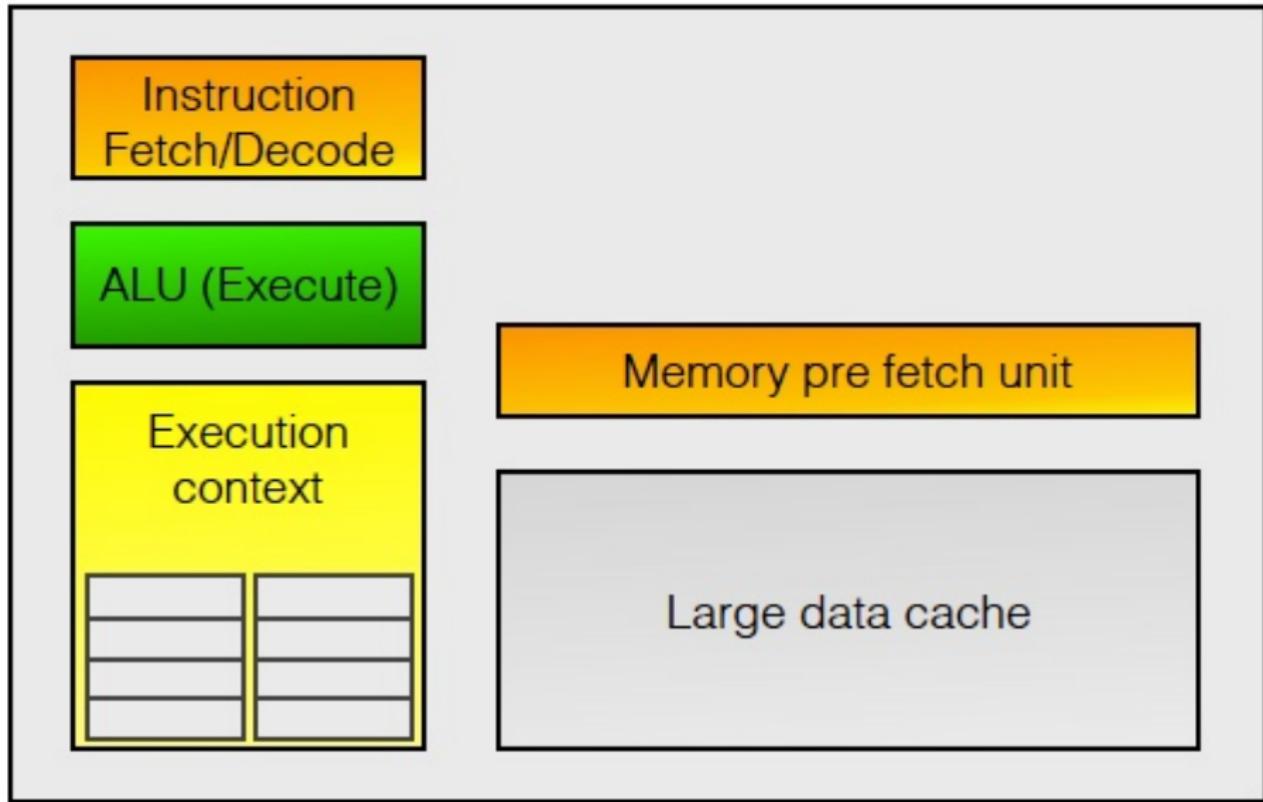
# GPU: Trim down CPU to core



# GPU: Trim down CPU to core



# GPU: Trim down CPU to core



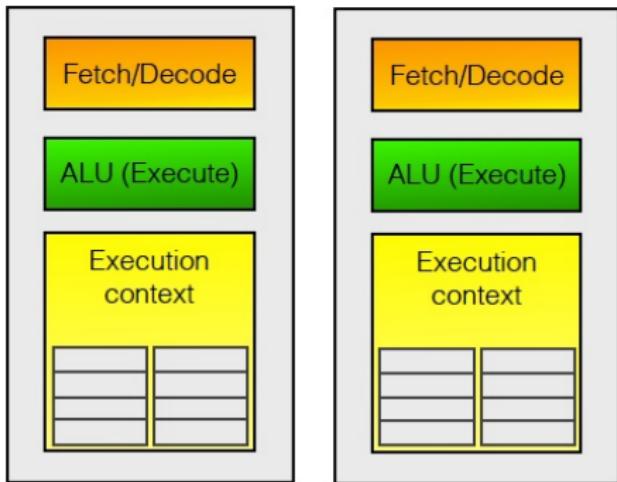
# GPU: Trim down CPU to core



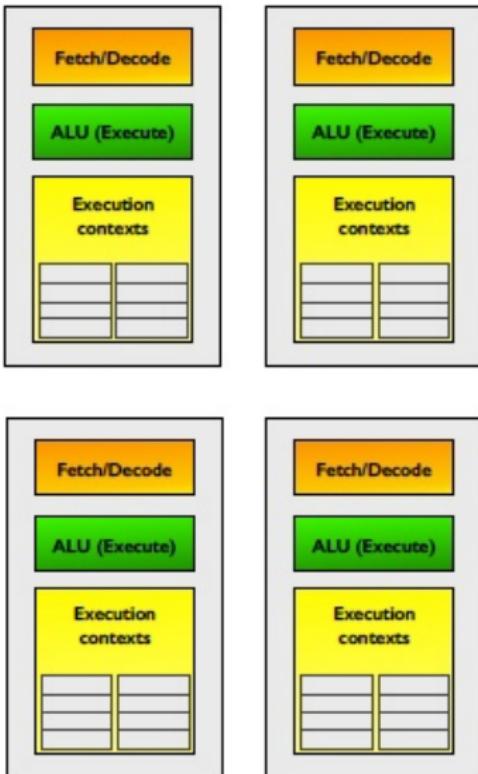
**Idea #1:**

**Remove the modules  
that help a single  
instruction execute fast.**

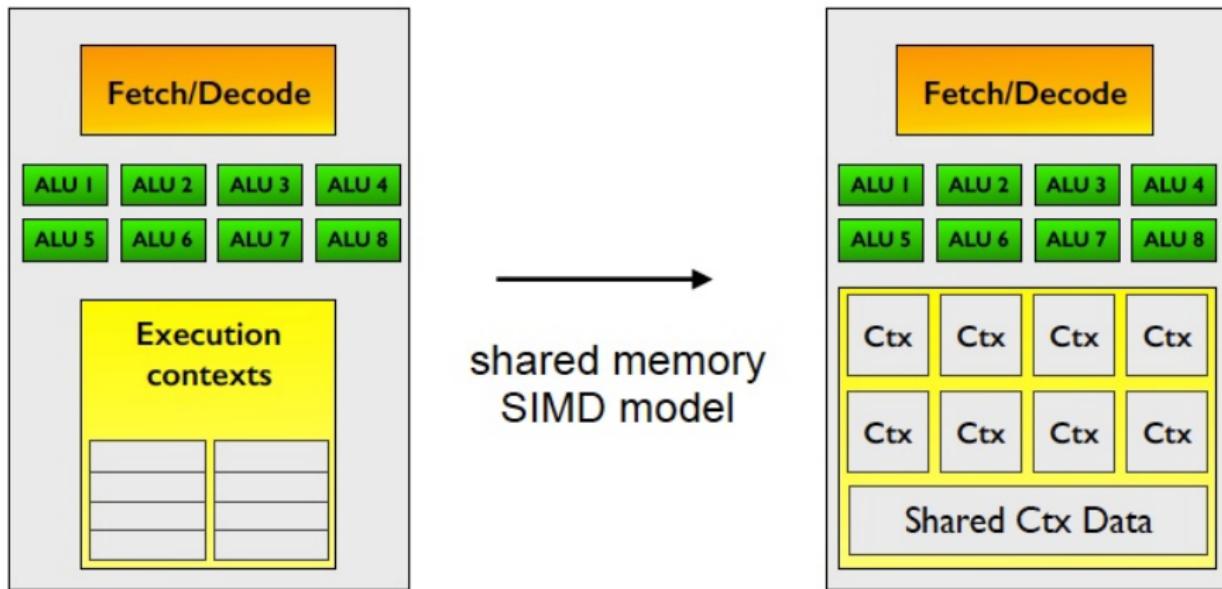
# GPU: Mimic core



# GPU: Mimic core



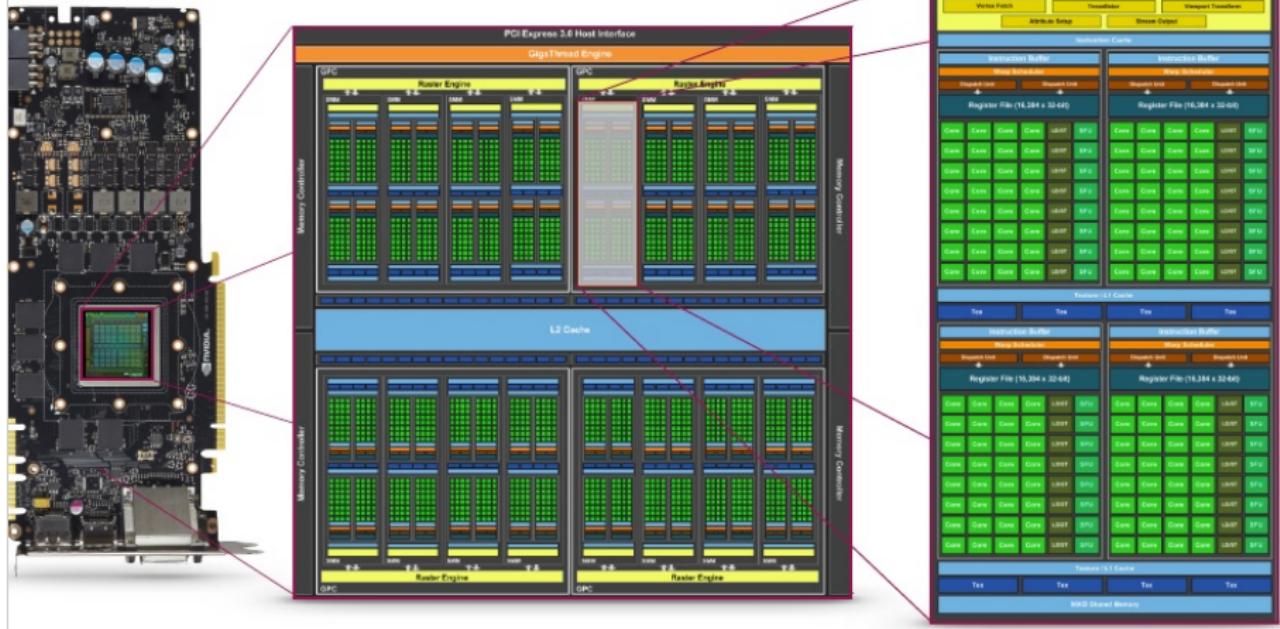
# GPU: SIMD + Shared Memory



# GPU: SIMD + Shared Memory



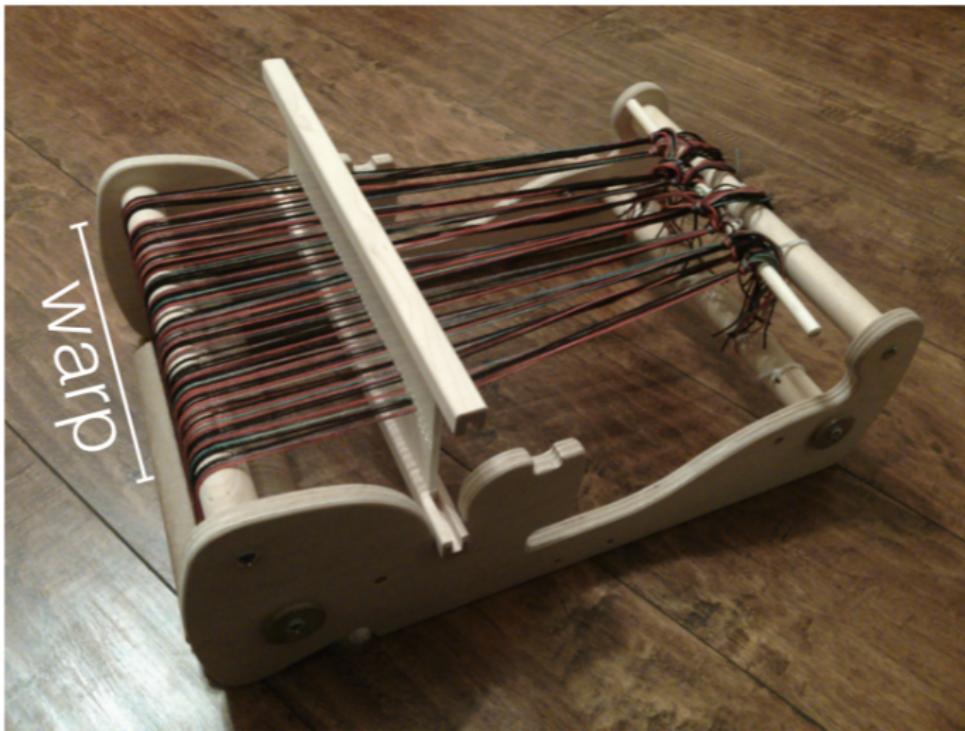
# GPU: NVIDIA Maxwell



16 Maxwell cores each have four SIMD clusters with 32 ALUs. Data streams at 56 GFLOAT/s and peak 4.6 TFLOP/s (SP)

# Terminology

derived from weaving; like warp, thread, texture.



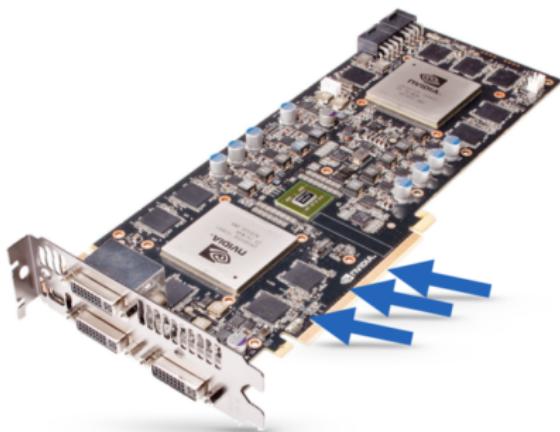
# CUDA- Offloading model

derived from weaving; like warp, thread, texture.



# CUDA- Offloading model

Explicitly moves data between Host and Device



1. cudaMalloc: allocate memory  
for a DEVICE array

2. cudaMemcpy: copy data  
from HOST to DEVICE array

# CUDA- Offloading model

Explicitly moves data between Host and Device



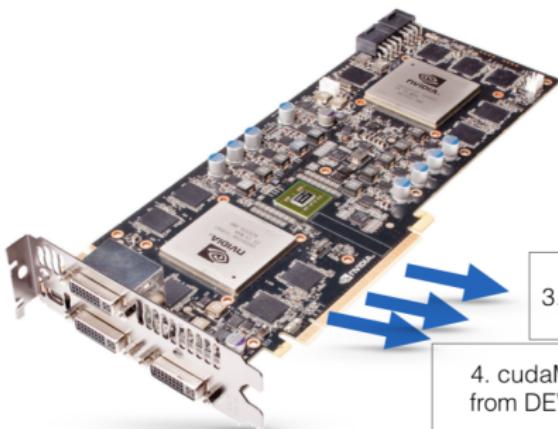
1. cudaMalloc: allocate memory  
for a DEVICE array

2. cudaMemcpy: copy data  
from HOST to DEVICE array

3. Queue kernel task on DEVICE

# CUDA- Offloading model

Explicitly moves data between Host and Device



1. cudaMalloc: allocate memory for a DEVICE array

2. cudaMemcpy: copy data from HOST to DEVICE array

3. Queue kernel task on DEVICE

4. cudaMemcpy: copy data from DEVICE to HOST array

# GPUs on midway2 and git repository

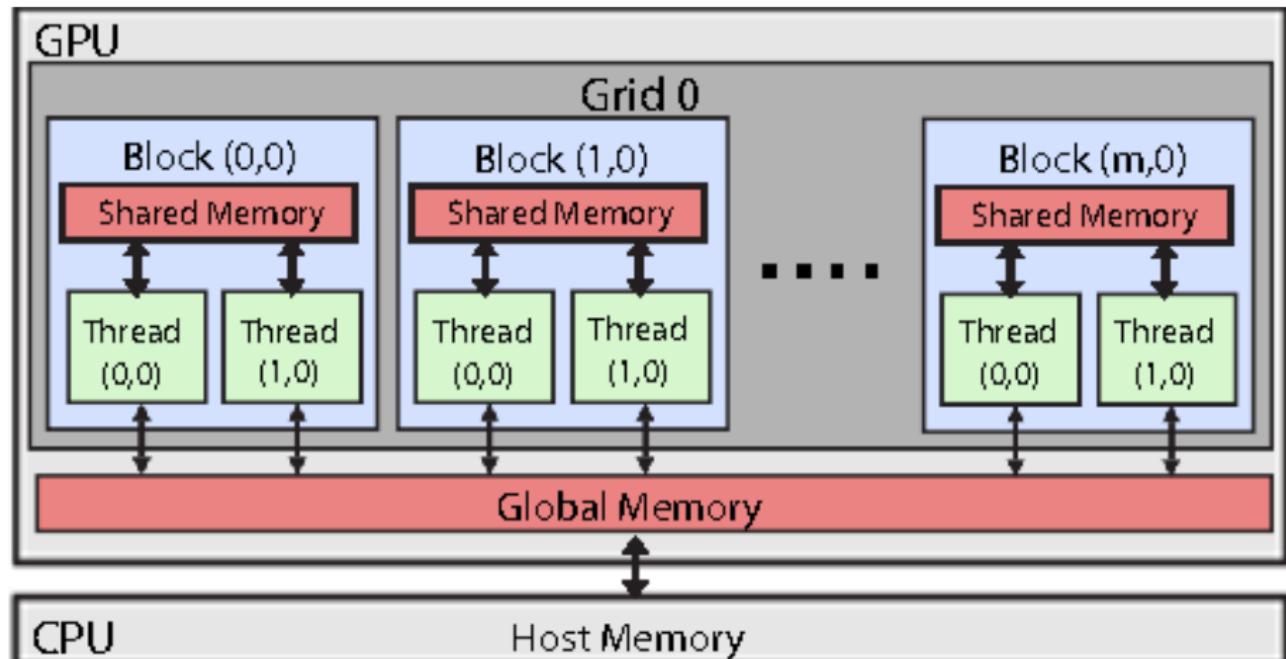
Allocate one gpu for the workshop with

```
sinteractive --partition=gpu2 --gres=gpu:1 --reservation=gpu-workshop
```

Fetch the git repository:

```
git clone https://github.com/rcc-uchicago/CUDA-C.git
```

# CUDA: Grid and Blocks



# CUDA: Host Code

```
int main(int argc, char **argv){  
    int N = 1024; // size of array for this DEMO  
  
    float *d_a; // Allocate DEVICE array  
    cudaMalloc((void**) &d_a, N*sizeof(float));  
  
    dim3 dimBlock(512,1,1); // 512 threads per thread-block  
    dim3 dimGrid((N+511)/512, 1, 1); // Enough thread-blocks to cover N  
  
    // Queue kernel on DEVICE  
    simpleKernel <<< dimGrid, dimBlock >>> (N, d_a);  
  
    // HOST array  
    float *h_a = (float*) calloc(N, sizeof(float));  
  
    // Transfer result from DEVICE to HOST  
    cudaMemcpy(h_a, d_a, N*sizeof(float), cudaMemcpyDeviceToHost);
```

# CUDA: host Code

- 1 Allocate array on device:

```
float *d_a; // Allocate DEVICE array (pointers used as array handles)
cudaMalloc((void**) &d_a, N*sizeof(float));
```

- 2 Allocate Threads and block:

```
dim3 dimBlock(512,1,1);           // 512 threads per thread-block
dim3 dimGrid((N+511)/512, 1, 1); // Enough thread-blocks to cover N
```

- 3 Queue up on device:

```
// specify number of threads with <<< block count, thread count >>>
SimpleKernel <<< dimGrid, dimBlock >>> (N, d_a);
```

- 4 Copy from device to host:

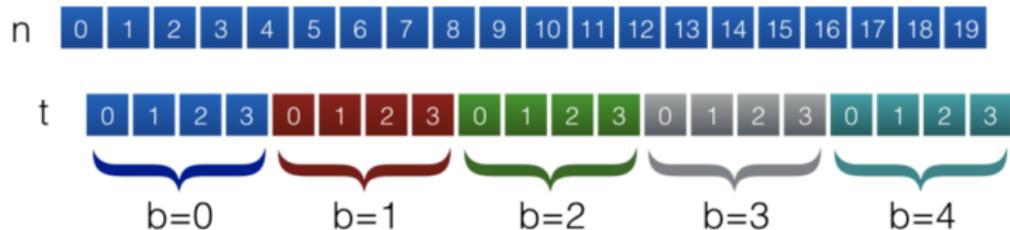
```
float *h_a = (float*) calloc(N, sizeof(float));
cudaMemcpy(h_a, d_a, N*sizeof(float), cudaMemcpyDeviceToHost)
```

# CUDA: A basic comprehension

```
void serialSimpleKernel(int N, float *d_a){  
    for(n=0;n<N;++n){ // loop over N entries  
        d_a[n] = n;  
    }  
}
```

# CUDA: A basic comprehension

```
void serialSimpleKernel(int N, float *d_a){  
    for(n=0;n<N;++n){ // loop over N entries  
        d_a[n] = n;  
    }  
}
```

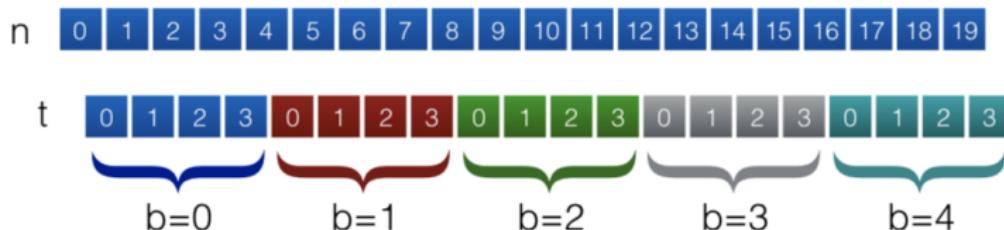


# CUDA: A basic comprehension

```
void tiledSerialSimpleKernel(int N, float *d_a){  
    for(int b=0;b<gridDim;++b){ // loop over blocks  
        for(int t=0;t<blockDim;++t){// loop inside block  
            // Convert thread and thread-block indices into array index  
            const int n = t + b*blockDim;  
            // If index is in [0,N-1] add entries  
            if(n<N) // guard against an inexact tiling  
                d_a[n] = n;  
        }  
    }  
}
```

# CUDA: A basic comprehension

```
void tiledSerialSimpleKernel(int N, float *d_a){  
    for(int b=0;b<gridDim;++b){ // loop over blocks  
        for(int t=0;t<blockDim;++t){// loop inside block  
            // Convert thread and thread-block indices into array index  
            const int n = t + b*blockDim;  
            // If index is in [0,N-1] add entries  
            if(n<N) // guard against an inexact tiling  
                d_a[n] = n;  
    }  
}
```



# CUDA: A basic comprehension

```
void tiledSerialSimpleKernel(int N, float *d_a){

    for(blockIdx.x=0;blockIdx.x<gridDim.x;++blockIdx.x){ // loop over blocks
        for(threadIdx.x=0;threadIdx.x<blockDim.x;++threadIdx.x){ // loop inside block

            // Convert thread and thread-block indices into array index
            const int n = threadIdx.x + blockDim.x*blockIdx.x;

            // If index is in [0,N-1] add entries
            if(n<N)
                d_a[n] = n;
        }
    }
}
```

# CUDA: A basic comprehension

```
void tiledSerialSimpleKernel(int N, float *d_a){  
    for(blockIdx.x=0;blockIdx.x<gridDim.x;++blockIdx.x){ // loop over blocks  
        for(threadIdx.x=0;threadIdx.x<blockDim.x;++threadIdx.x){ // loop inside block  
            // Convert thread and thread-block indices into array index  
            const int n = threadIdx.x + blockDim.x*blockIdx.x;  
  
            // If index is in [0,N-1] add entries  
            if(n<N)  
                d_a[n] = n;  
    }  
}
```

# CUDA: A basic comprehension for FORTRAN

```
module mathOps
contains
    attributes(global) subroutine saxpy(x, y, a)
        implicit none
        real :: x(:), y(:)
        real, value :: a
        integer :: i, n
        n = size(x)
        i = blockDim*x * (blockIdx*x - 1) + threadIdx*x
        if (i <= n) then
            y(i) = y(i) + a*x(i)
        end if
    end subroutine saxpy
end module mathOps

program testSaxpy
    use mathOps
    use cudafor
    implicit none
    integer, parameter :: N = 40000
    real :: x(N), y(N), a
    real, device :: x_d(N), y_d(N)
    type(dim3) :: grid, tBlock

    tBlock = dim3(256,1,1)
    grid = dim3(ceiling(real(N)/tBlock*x),1,1)
    print *, 'Number of threads in each block:', tBlock*x
    print *, 'Number of Blocks:', grid*x
    x = 1.0; y = 2.0; a = 2.0
    x_d = x
    y_d = y
    call saxpy<<<grid, tBlock>>>(x_d, y_d, a)
    y = y_d
    write(*,*) 'Max error: ', maxval(abs(y-4.0))
end program testSaxpy
```

# CUDA: Block and thread structure

Intrinsic variables			
Description	Fastest index		Slowest index
Thread indices in thread-block	threadIdx.x	threadIdx.y	threadIdx.z
Dimensions of thread-block	blockDim.x	blockDim.y	blockDim.z
Block indices.	blockIdx.x	blockIdx.y	blockIdx.z *
Dimensions of grid of thread-blocks	gridDim.x	gridDim.y	gridDim.z *

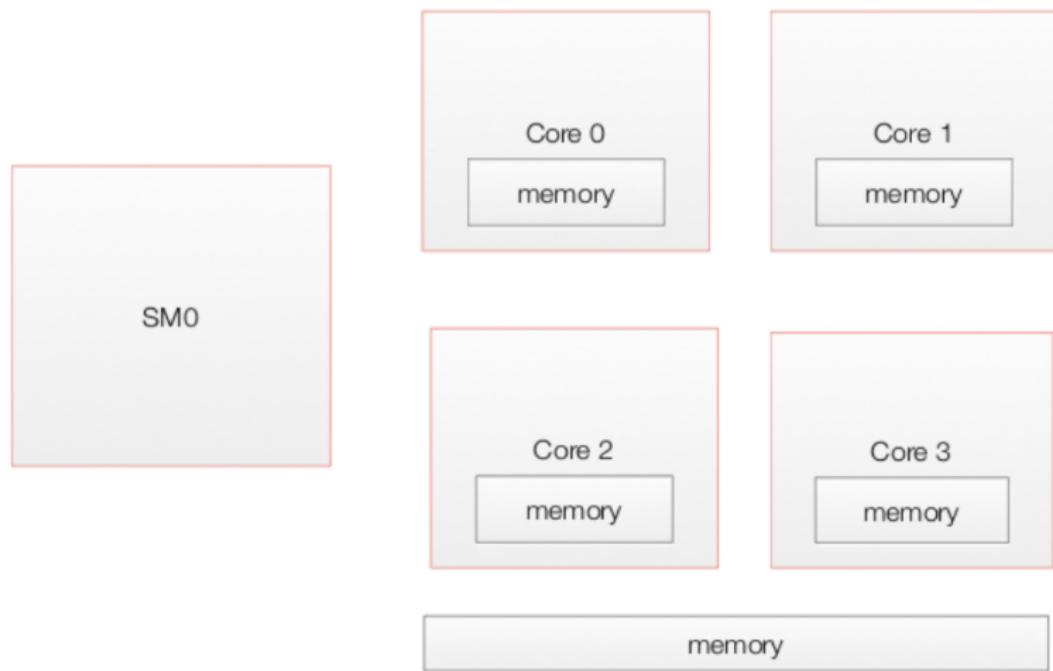
# CUDA: Limitations

Technical specifications	Compute capability (version)																												
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0 (7.2?)	7.5												
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.			16				4				32				16		128											
Maximum dimensionality of grid of thread blocks	2											3																	
Maximum x-dimension of a grid of thread blocks	65535												$2^{31} - 1$																
Maximum y-, or z-dimension of a grid of thread blocks	65535																												
Maximum dimensionality of thread block	3																												
Maximum x- or y-dimension of a block	512											1024																	
Maximum z-dimension of a block	64																												
Maximum number of threads per block	512											1024																	
Warp size	32																												
Maximum number of resident blocks per multiprocessor	8			16								32						16											
Maximum number of resident warps per multiprocessor	24	32	48									64						32											
Maximum number of resident threads per multiprocessor	768	1024	1536									2048						1024											
Number of 32-bit registers per multiprocessor	8 K	16 K	32 K	64 K		128 K						64 K																	
Maximum number of 32-bit registers per thread block	N/A			32 K		64 K		32 K		64 K		32 K		64 K		32 K		64 K											
Maximum number of 32-bit registers per thread	124			63															255										
Maximum amount of shared memory per multiprocessor	16 KB			48 KB				112 KB		64 KB		96 KB		64 KB		96 KB		64 KB		96 KB (of 128)									
Maximum amount of shared memory per thread block	48 KB																	48/96 KB		64 KB									
Number of shared memory banks	16																	32											

# A basic optimization: shared memory approach



# A basic optimization: shared memory approach



# Performance: Roof-line model for GPU

## Tesla K10 Peak Single Precision FLOPs:

745 MHz core clock \* 2 GPUs/board \*(8 multiprocessors \* 192 fp32 cores/multiprocessor) \* 2ops/cyle=4.58 TFLOPs.

## Tesla K10 Memory Bandwidth FLOPs:

2 GPUs/board \* 256 bit \* 2500 MHz mem-clock \* 2 DDR / 8 bits/byte = 320 GB/s

Ratio of instruction : bytes:

4.58 TFLOPS / 320 GB/s = 13.6 instructions : 1 byte

# Acknowledgment

- Prof. Jesse Chan, Rice University, Houston, TX
- Prof. Tim Warburton, Virginia Tech, VA

Thank You! Questions?