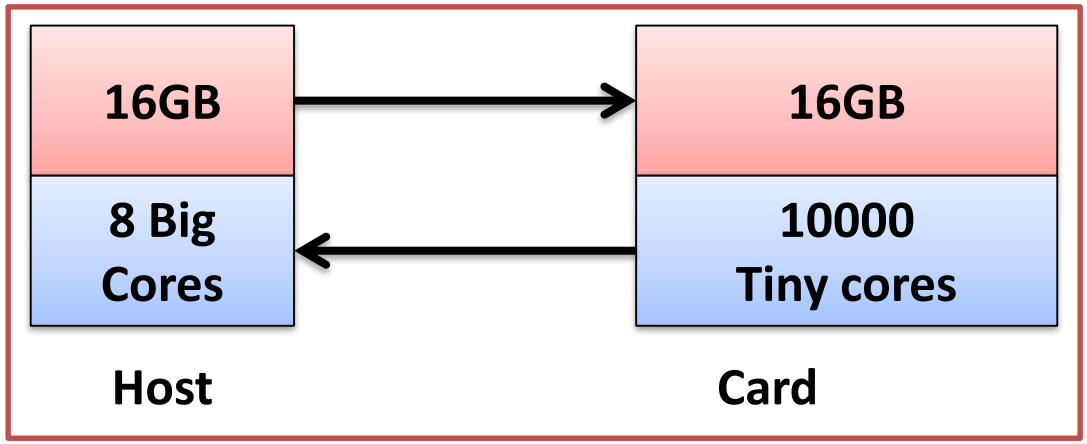
GPU
&
Application Analysis for GPGPU

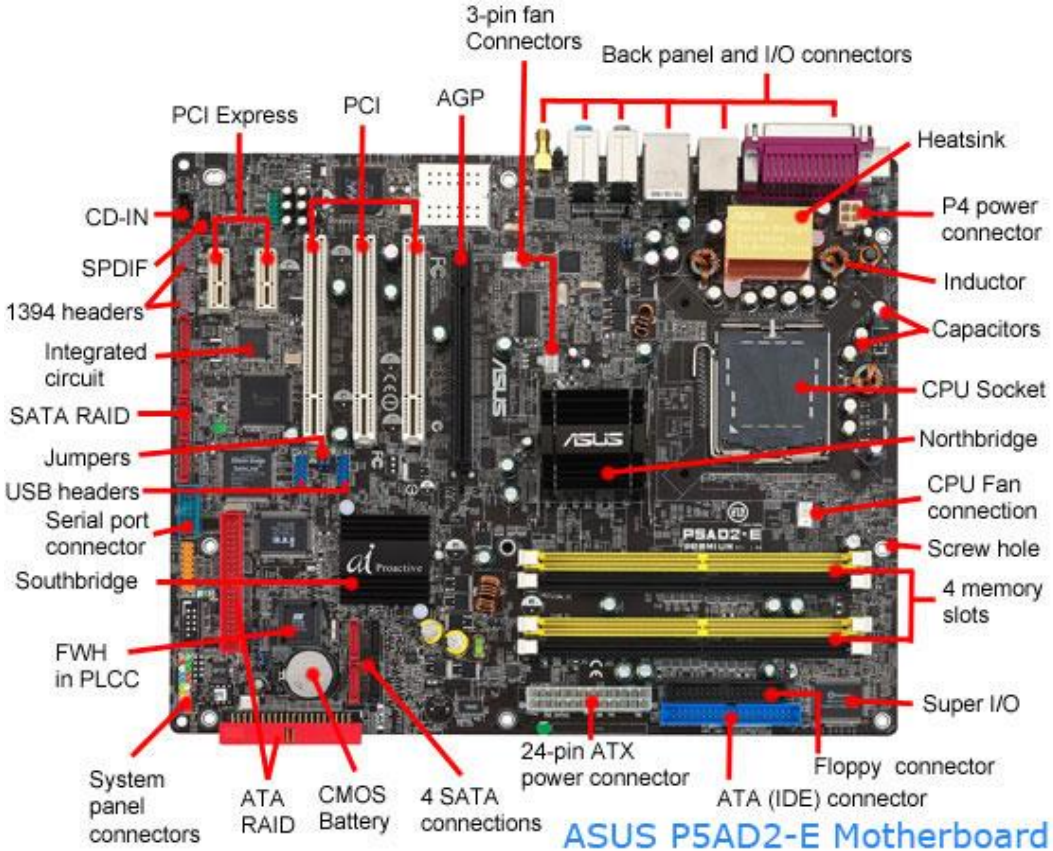
A. Sahu

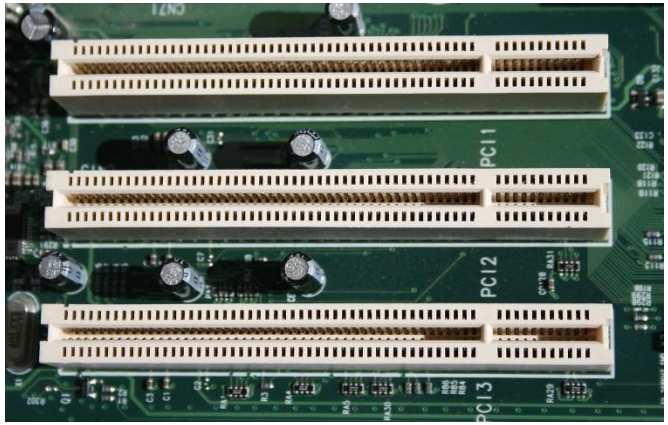
Dept of CSE, IIT Guwahati

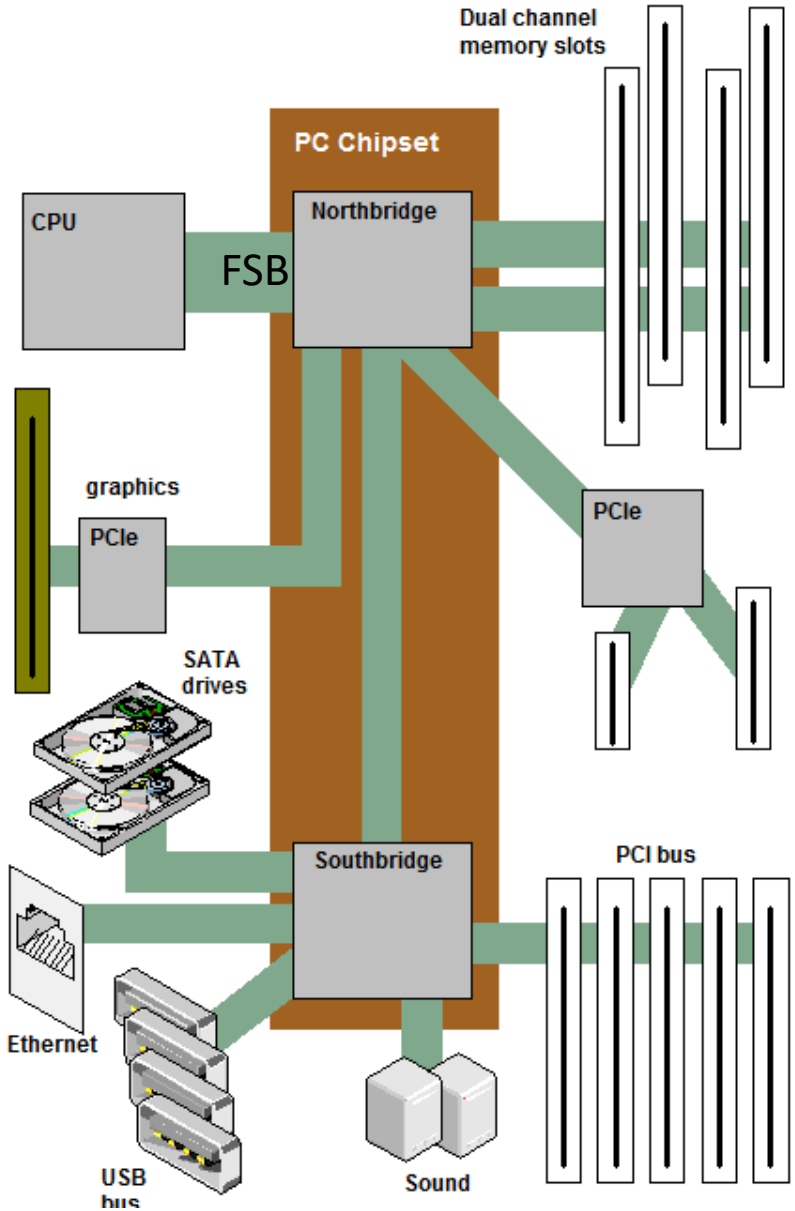
GPU

- Graphics Cards to Motherboard PCI Slot
  - Peripheral Components Interconnect
- To accelerate Graphics computation
- Earlier day : It was fixed purpose
- Now a days, it is programmable, configurable
  - Why not to use them for general purpose?
  - For what kind of application



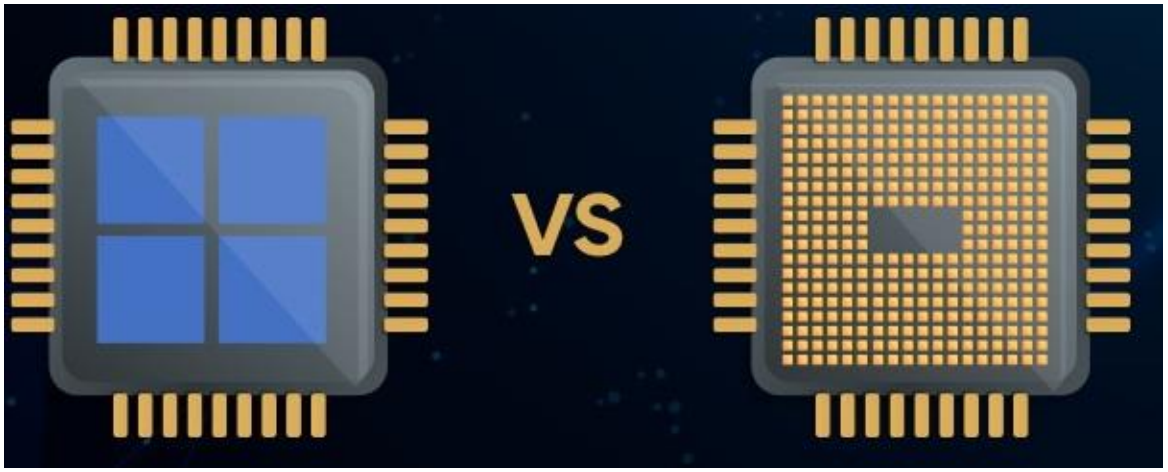

3-pin fan Connectors
Back panel and I/O connectors
PCI Express
PCI
AGP
Heatsink
CD-IN
P4 power connector
SPDIF
Inductor
1394 headers
Capacitors
Integrated circuit
CPU Socket
SATA RAID
Northbridge
Jumpers
USB headers
CPU Fan connection
Serial port connector
Screw hole
Southbridge
4 memory slots
FWH in PLCC
Super I/O
System panel connectors
ATA RAID
CMOS Battery
4 SATA connections
24-pin ATX power connector
ATA (IDE) connector
Floppy connector
ASUS P5AD2-E Motherboard
ComputerHope.com




Dual channel memory slots
PC Chipset
CPU
FSB
Northbridge
graphics
PCIe
PCIe
SATA drives
Southbridge
PCI bus
Ethernet
USB bus
Sound

GPU

- GPU vs CPU



- GPU Cards
  - GTX3090: **10496** cores, 24GB DDR6-384bit interface, Rs 1.3L

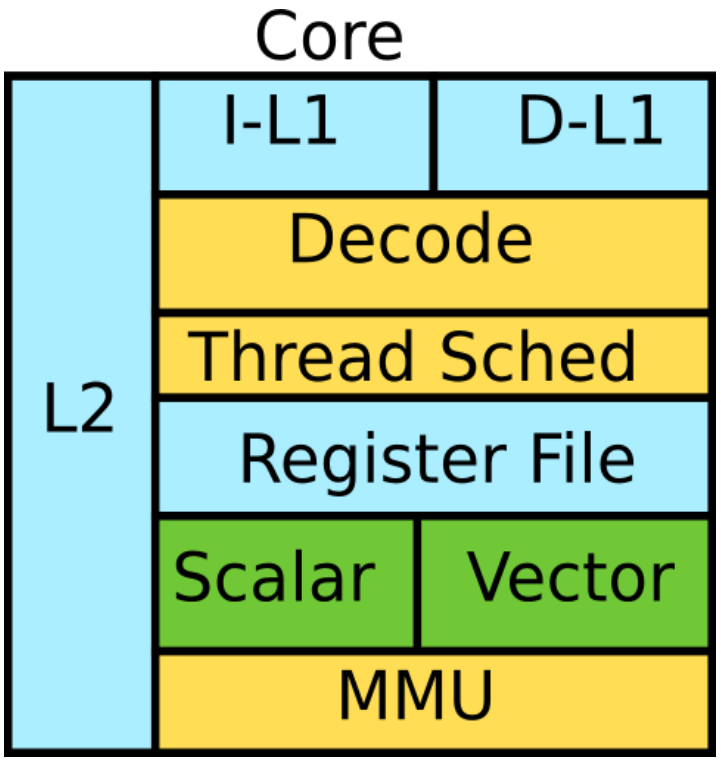
GPU Philosophy

- **Small independent function/code executed huge number of time**
- **Number of cores in thousands, tiny cores**
- Cores are organized in cluster
  - Kepler SMX: 14 SM, 192 SP Cuda cores/SM, 64 DP units, 32 SFU, 32 (LD/ST) U
  - TU102-RTX 2080Ti: 72 SM, 4608 Cuda Cores, 576 Tensor core, 72 RayTrace cores
- Explicit memory hierarchy, programmer controlled
- Also implicit memory hierarchy : cache

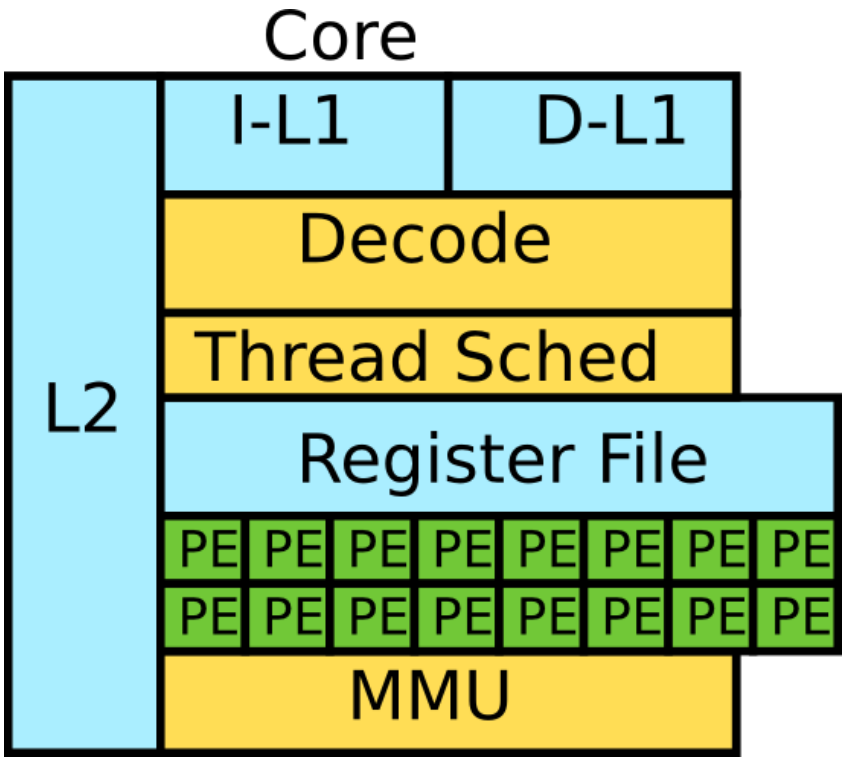
Single Instruction Multiple Data

GPU uses wide SIMD: 8/16/24/... processing elements (PEs)
CPU uses short SIMD: usually has vector width of 4/8.

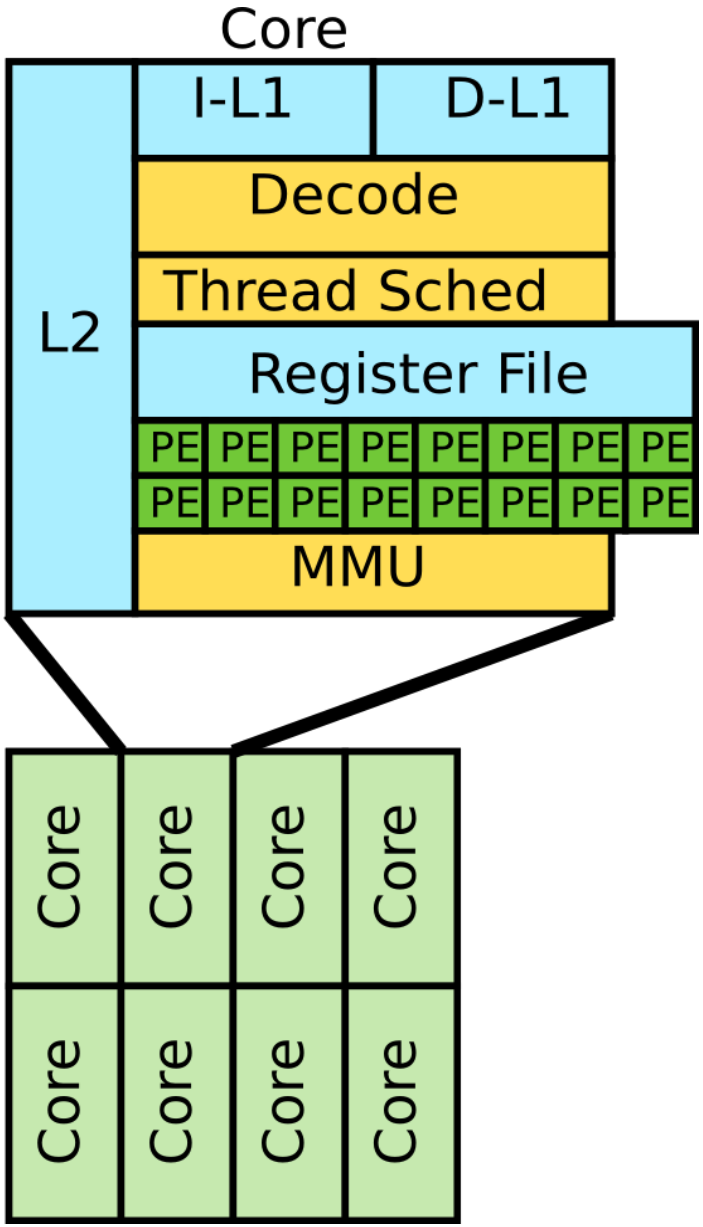
SSE has 4 data lanes



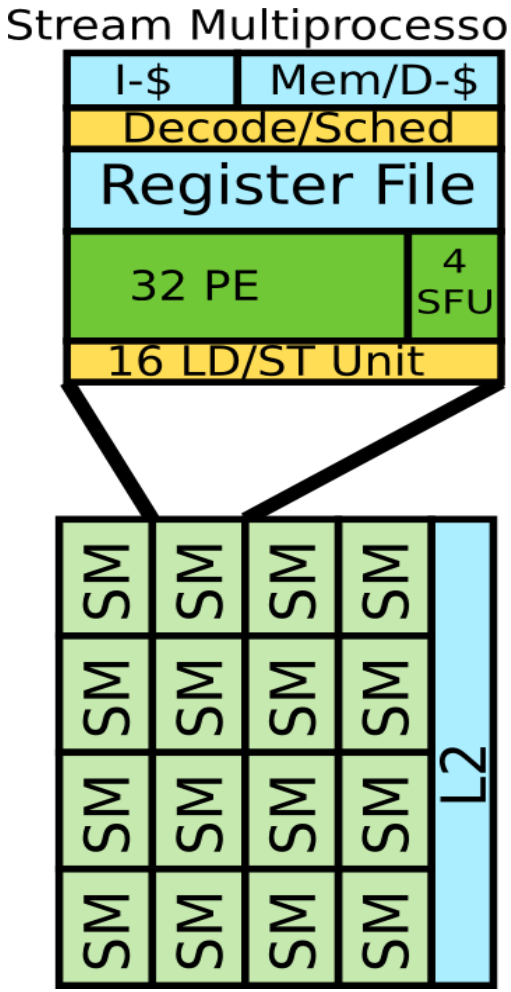
GPU has 8/16/24/... data lanes



Simple Core



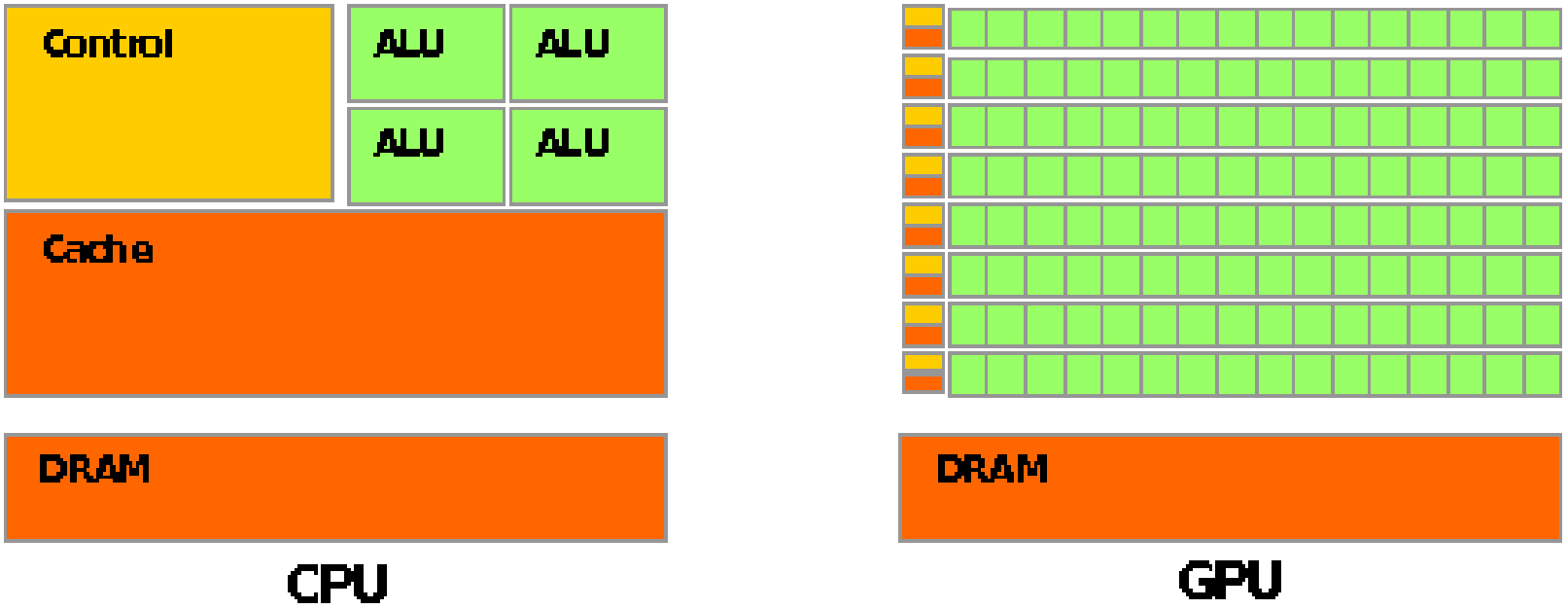
The Stream Multiprocessor (SM) is a light weight core compared to IA core.



Light weight PE:
Fused Multiply Add (FMA)

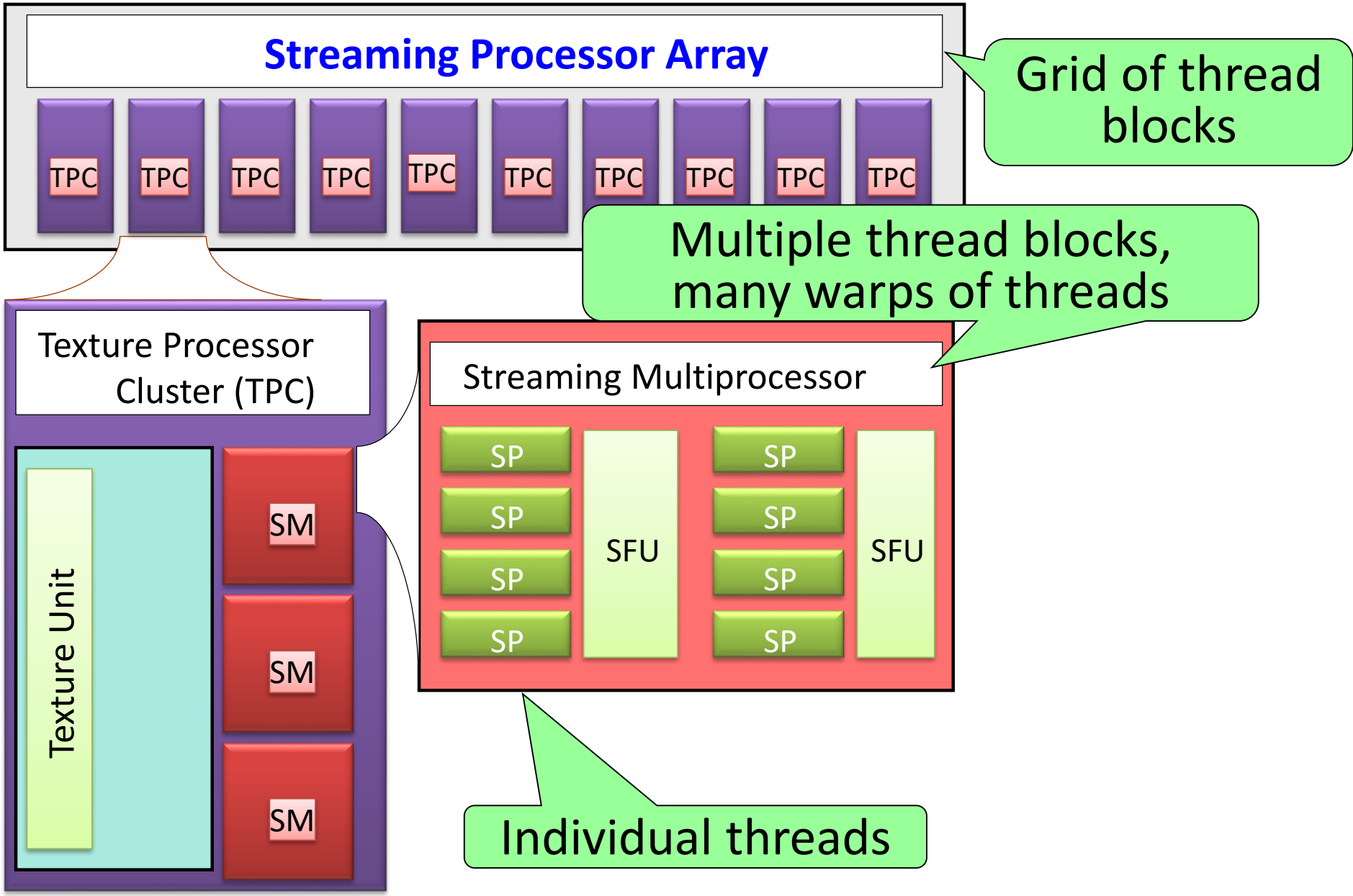
SFU:
Special Function Unit

CPU vs GPU Layout



Source: Nvidia CUDA Programming Guide

NVIDIA architecture



GTX 690-Architetcure

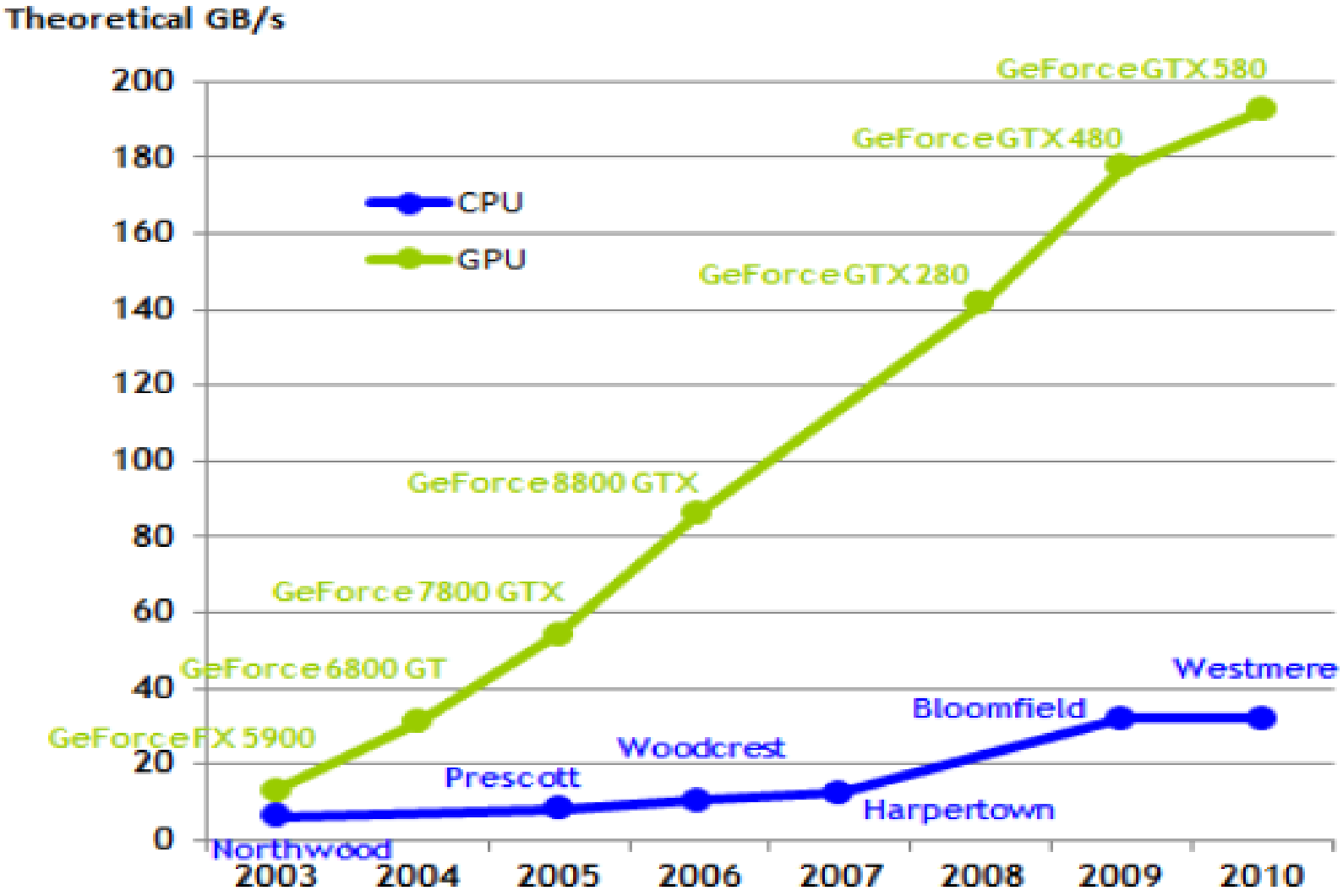
GeForce GTX 980Ti Technical Specs.

- 2816 cuda cores
- 22 SMX, 128 SP/SMX
- 6GB DDR5 RAM
- Each SMX have
  - Texture Cache 56KB
  - Constant memory (Scratch PAD) 65 KB
  - 49 KB L1/shared memory
  - Uniform Cache
  - Separate Shares Instruction Cache for SM

GeForce RTX 2080Ti Technical Specs.

- 4606 cuda cores
- 72 SMX, 64 SP/SMX
- **576 Tensor Core**
- **72 RayTrace Core**
- 11GB DDR6 RAM
- Each SMX have
  - Texture Cache 56KB
  - Constant memory (Scratch PAD) 65 KB
  - 49 KB L1/shared memory
  - Uniform Cache
  - Separate Shares Instruction Cache for SM

CPU-GPU Performance Gap



Source: CUDA Prog. Guide 4.0

Graphical Processing Units

- Given the hardware invested to do graphics well, how can be supplement it to improve performance of a wider range of applications?
- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Prog. model is "**Single Instruction Multiple Thread**"

Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks
- Blocks are organized into a grid
- GPU hardware handles thread management, not applications or OS

Mapping SW Organized to GPU

- Software : Cuda Program or Cuda Threads
  - Thread organized into Grid, Block and Threads
  - Grid contain Blocks, Block contains many threads
- Hardware: GPU
  - Cores are organized in to clusters (SM)
  - GTX 690 per device: 8 SM, 192 core/SM
  - GTX980Ti: 22 SM, 128core/SM
- Mapping
  - Block get mapped to SM
  - Thread get map to core (SP)

Example thread scheduling

- Suppose we want to create parallel 2000 threads for an application
- We organize the thread into 10 blocks, each contains 200 threads
- When we run on top of GTX 690 GPU with 8 SM with 192 SP/SM
- Scheduler:
  - Map 10 blocks to 8 SM, takes ceil(10/8) =2 times
  - Map 200 threads to 192 SP of SM, it also takes ceil(200/192)=2 times
- **Thumb rules:**
  - Num block should be multiple of SM
  - Num thread/Block should be multiple of SP/SM

Example: DAXPY in C

```
//Invoke DAXPY
DAXPY(n,2.0,x,y);
//function in C
void DAXPY(int n, double a,
           double *x, double *y){
    for(int i=0;i<n;i++)
        y[i]=a*x[i]+y[i];
}
```

DAXPY in C and Cuda

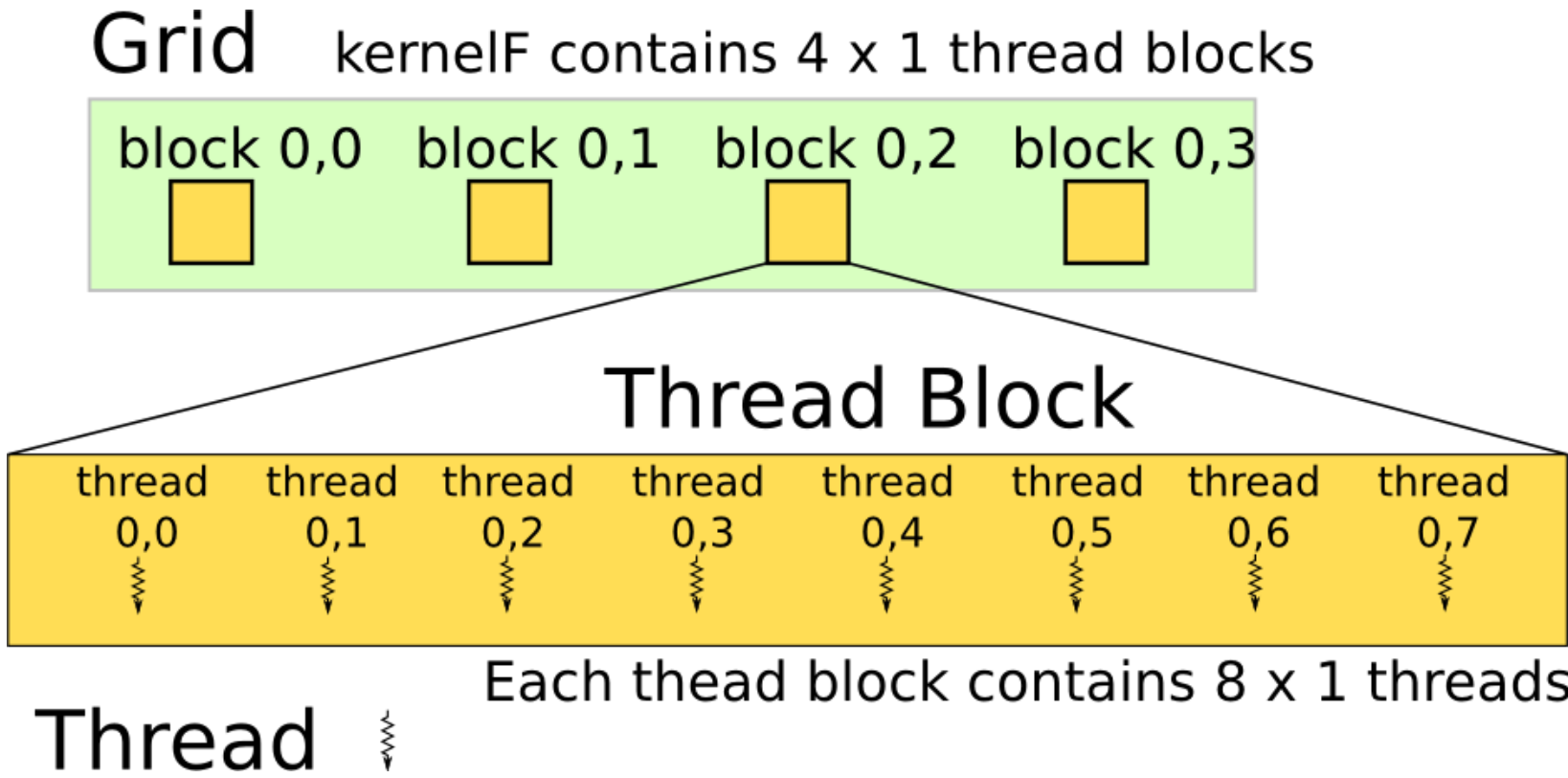
```
//Invoke DAXPY with
//256 threads per threadblock
__host__ int nb=(n+255)/256;
DAXPY<<<nb,256>>>(n,2.0,x,y);
//function in Cuda
__device__ void DAXPY(int n, double a,
                      double *x, double *y){
  int  i=blockIdx.x*blockDim.x
                    +threadIdx.x;
  if (i<n)
        y[i]=a*x[i]+y[i];
}
```

Complete DAXPY in Cuda

```
__device__  void DAXPY(int n, double a, double *x, double *y){
        int  i;
        i=blockIdx.x*blockDim.x +threadIdx.x;
        If (i<n)  y[i]=a*x[i]+y[i];
}
main(){                                          #define N 1024
        int x[N], y[N], size=sizeof(int)*N;      Initialize(x,y);
        __device__ int  *x_d, *y_d;  //Declaration and Memory  Creation
        cudaMalloc( (void **)&x_d, size);   // On device
        cudaMalloc( (void **)&y_d, size);
        cudaMemcpy( x_d, x, size, cudaMemcpyHostToDevice );
        cudaMemcpy( y_d, y, size, cudaMemcpyHostToDevice );
        __host__ int nb=ceil(N/256.0); //Invoke DAXPY with  256 thrds/TB
        DAXPY<<<nb,256>>>(N,2.0,xd,yd);
        cudaMemcpy( y, y_d, size, cudaMemcpyDeviceToHost );
}
```

Two Levels of Thread Hierarchy

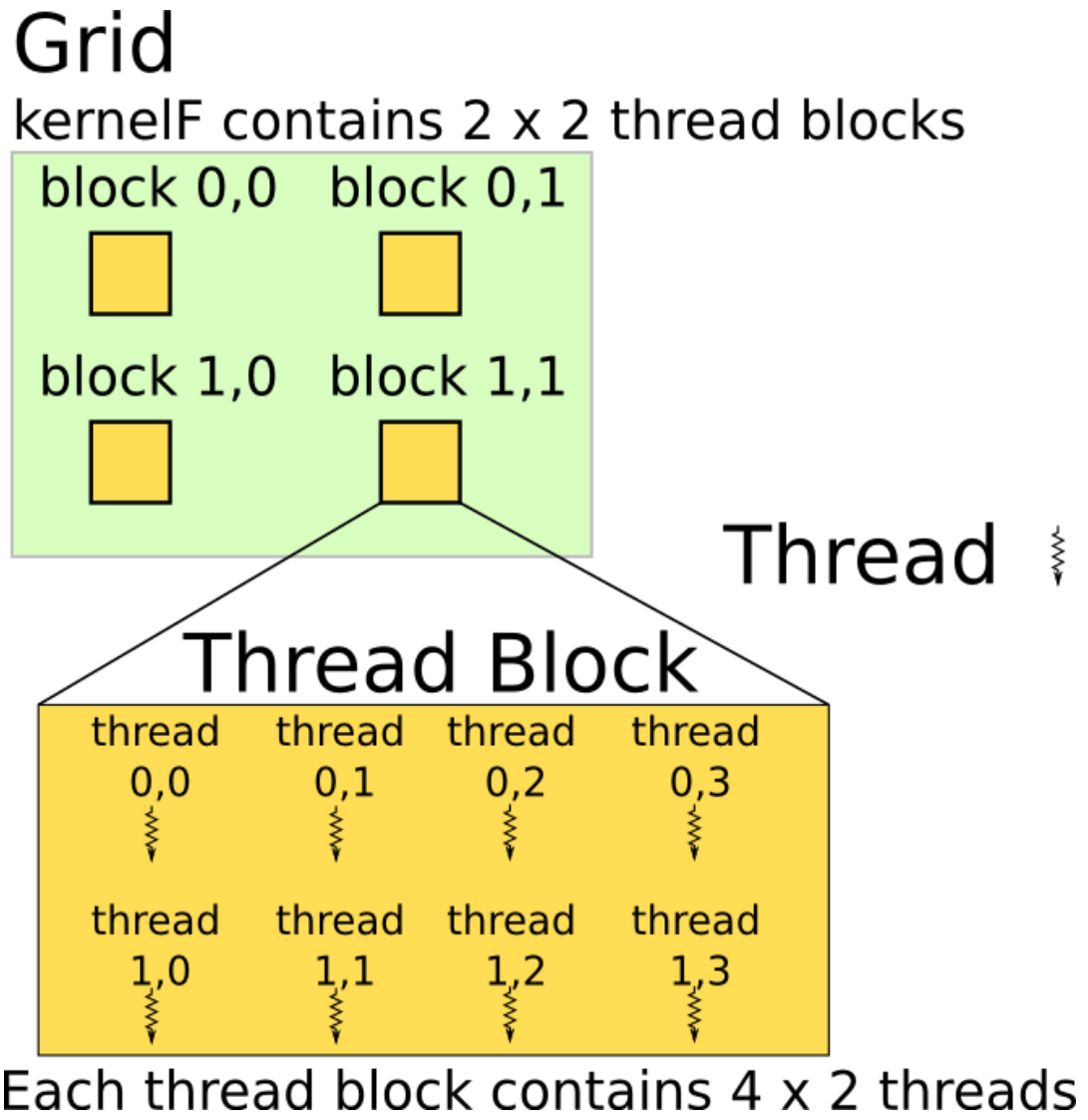
```
kernelF<<<(4,1),(8,1)>>>(A);

__device__    kernelF(A){
    i = blockIdx.x;
    j = threadIdx.x;
    A[i][j]++;
}
```



Multi-dimension Thread and Block ID

Both grid and thread block can have two dimensional index.

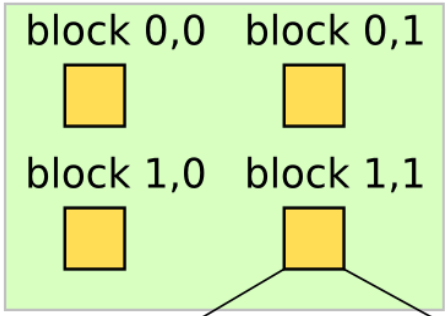
```
kernelF<<<(2,2),(4,2)>>>(A);

__device__     kernelF(A){
    i = blockDim.x * blockIdx.y
        + blockIdx.x;
    j = threadDim.x * threadIdx.y
        + threadIdx.x;
    A[i][j]++;
}
```

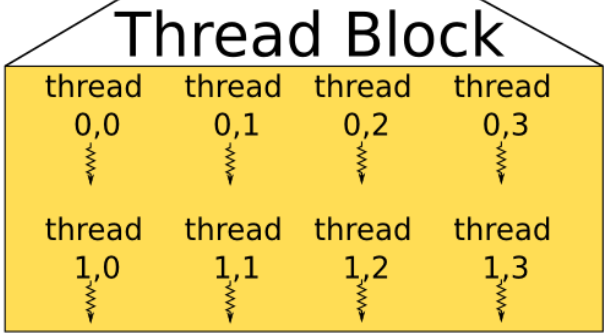


Scheduling Thread Blocks on SM

Grid

kernelF contains 2 x 2 thread blocks

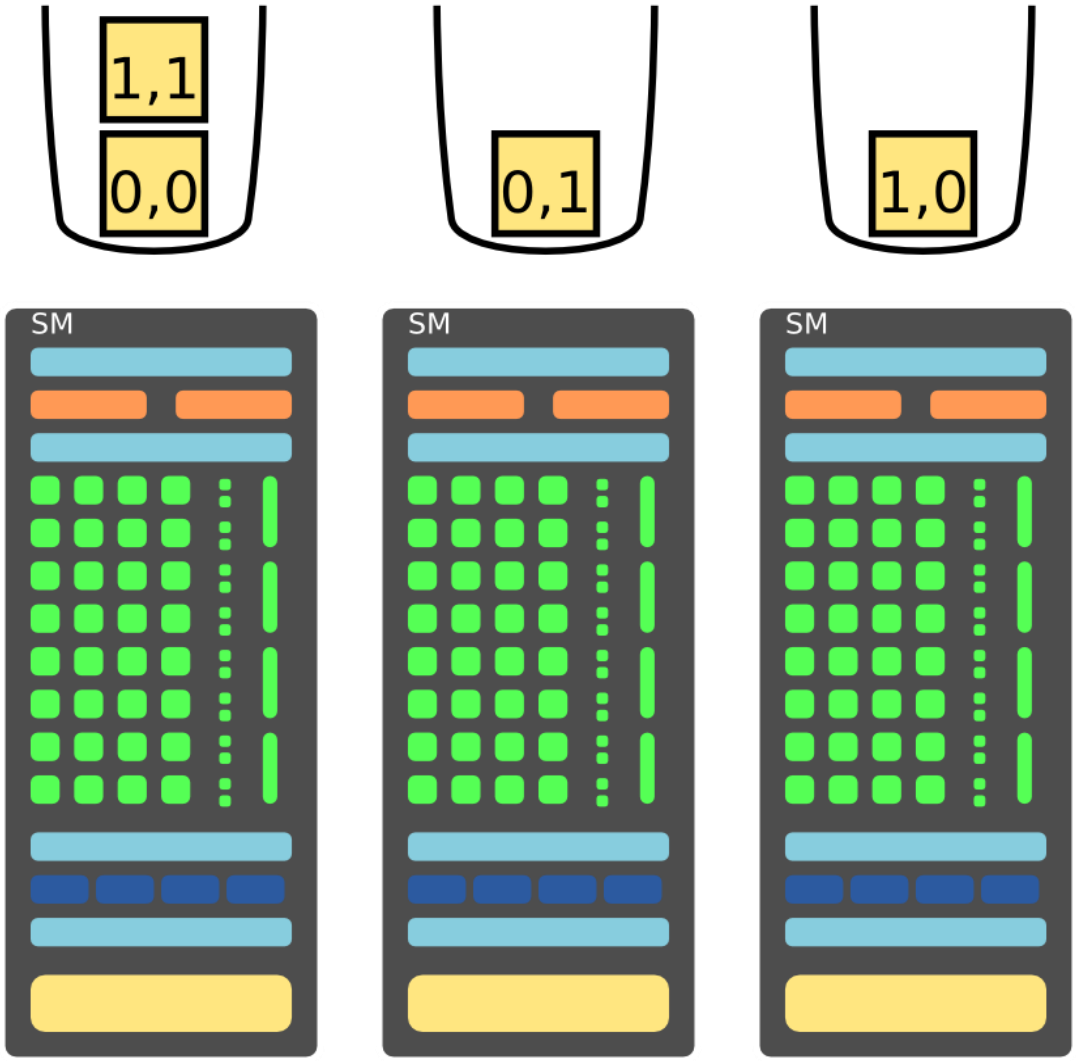




Each thread block contains 4 x 2 threads

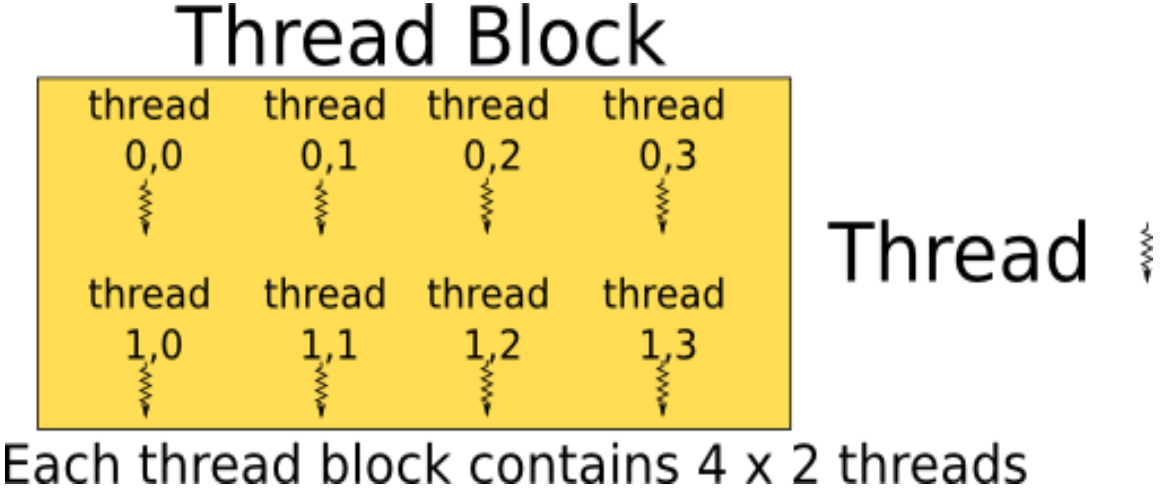
Example:

Scheduling 4 thread blocks on 3 SMs.



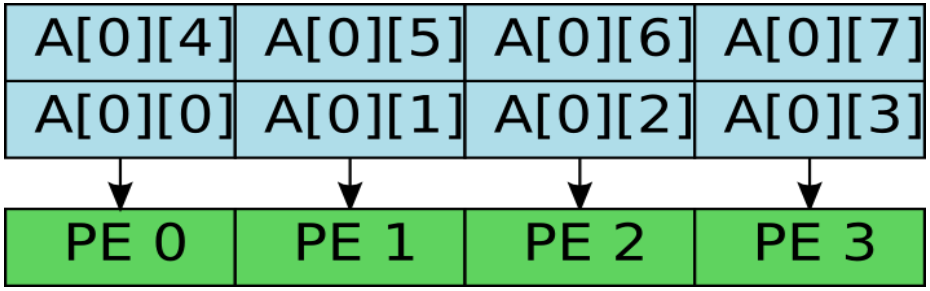
Executing Thread Block on SM

```
kernelF<<<(2,2),(4,2)>>>(A);

__device__    kernelF(A){
    i = blockDim.x * blockIdx.y
        + blockIdx.x;
    j = threadDim.x * threadIdx.y
        + threadIdx.x;
    A[i][j]++;
}
```

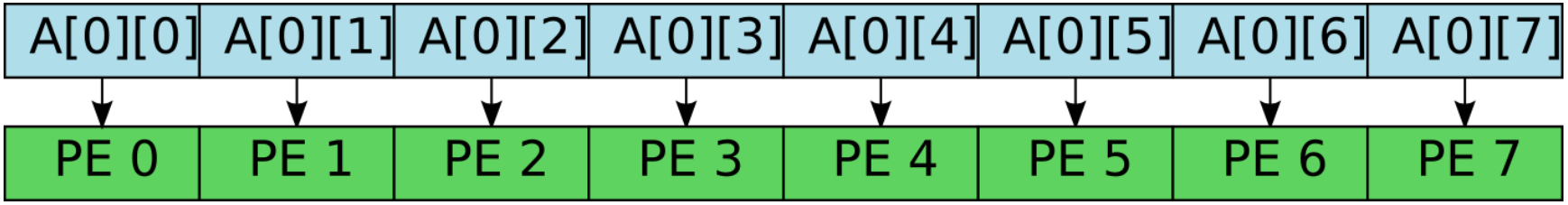


Notes: the number of Processing Elements (PEs) is transparent to programmer.

Executed on machine with width of 4:



Executed on machine with width of 8:

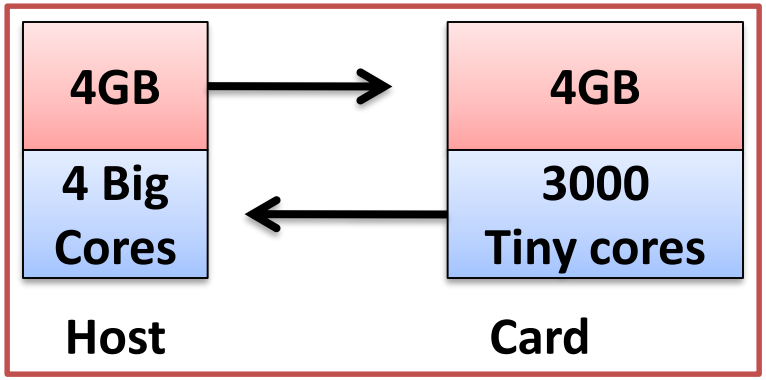


Examples: Application Mapping to Accelerator

- Machine Configuration
  - Pentium Quad core with 4GB RAM and installed GPU card with 3000 core and 4GB RAM on GPU
  - Need to consider Host to GPU memory data transfer and vice-verse.
- Applications
  - Vector Addition, Vector Sum
  - Matrix Multiplication
  - N-Body Simulation
  - Image Adaptive Histogram Equalization

Example 1: Vector Addition

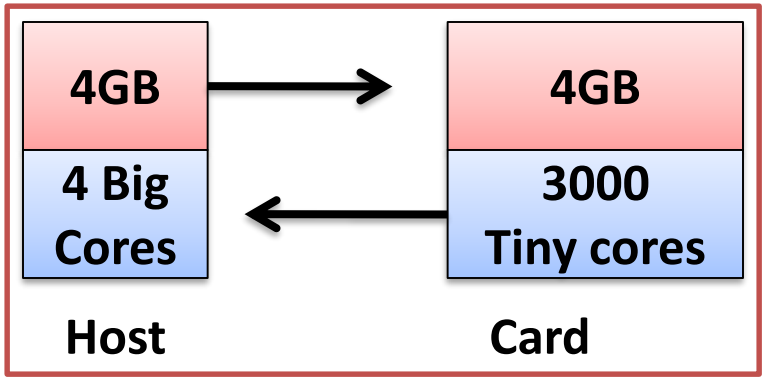
```
int A[1000], B[1000], C[1000];
for(i=0;i<1000;i++)
//Parallel, Independent Work
        A[i]=B[i]+C[i];
```



- Executing on Host
  - O(n) time, zero data transferred

Example 1: Vector Addition

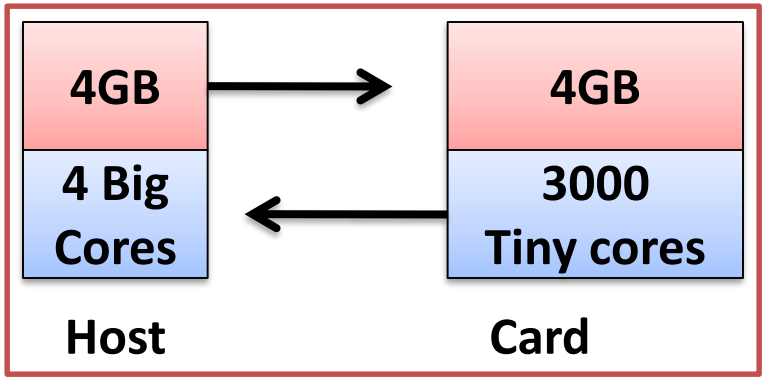
```
int A[1000], B[1000], C[1000];
for(i=0;i<1000;i++)
//Parallel, Independent Work
        A[i]=B[i]+C[i];
```



- Executing on GPU
  - 2*1000 data need to send from Host to GPU
  - Execution in O(1) time parallel using 1000 cores
  - 1000 data need to be return from GPU to Host
- Take: 3000 unit communication overheads funny, This **program is not a good candidate to run on GPU**

Example 1: Vector Addition

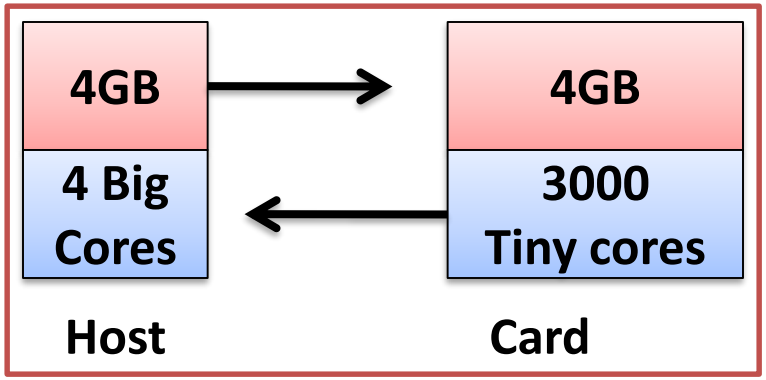
```
int A[1000], B[1000], C[1000];
for(i=0;i<1000;i++)
//Parallel, Independent Work
        A[i]=B[i]+C[i];
```



- Executing on the Host
  - using OpenMP/Pthread
  - Each thread can run from 250 locations
  - No need to transfer data, shared among all...
  - **Good news : (N/4 time)**

Example 2: Vector Sum

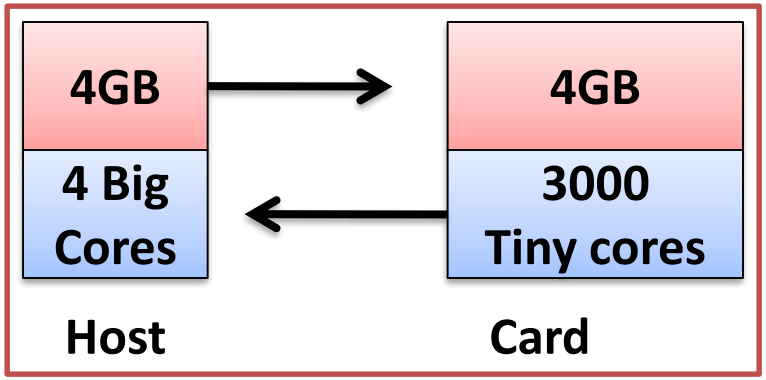
```
int A[1000];
for(i=0;i<1000;i++)
//Parallel, Independent Work
        S= S+A[i];
```



- Executing on Host
  - O(n) time, zero data transferred

Example 2: Vector Sum

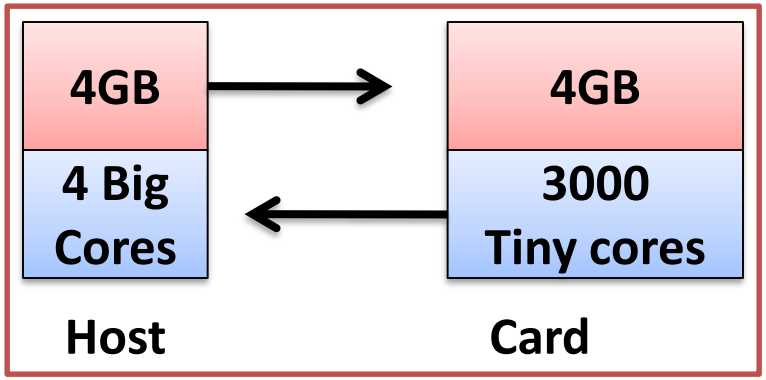
```
int A[1000];
for(i=0;i<1000;i++)
//Parallel, Independent Work
        S= S+A[i];
```



- Executing on GPU
  - 1000 data need to send from Host to GPU
  - Execution in Lg(1000) time parallel using 1000 cores **(GPU core is have very week support for shared variable locking)**
  - 1 data need to be return from GPU to Host
- Take: 1000 unit communication overheads funny, This **program is not a good candidate to run on GPU**

Example 2: Vector Sum

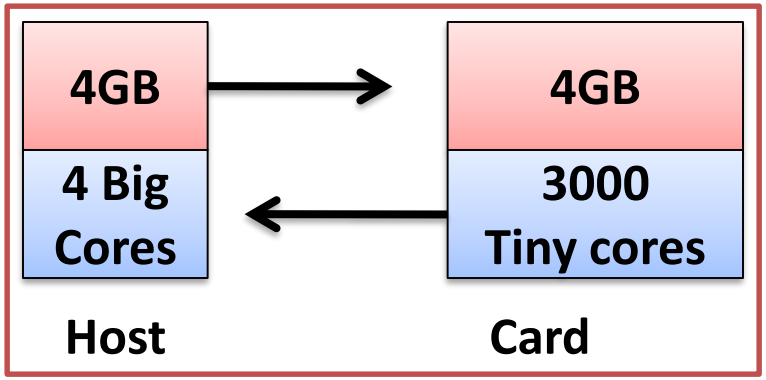
```
int A[1000];
for(i=0;i<1000;i++)
//Parallel, Independent Work
        S= S+A[i];
```



- Executing on the Host
  - Using OpenMP/Pthread
  - Each thread can run the sum from 250 locations
  - No need to transfer data, shared among all...
  - **Good news : time (N/4 +4)**

Example 3: Matrix Multiplication

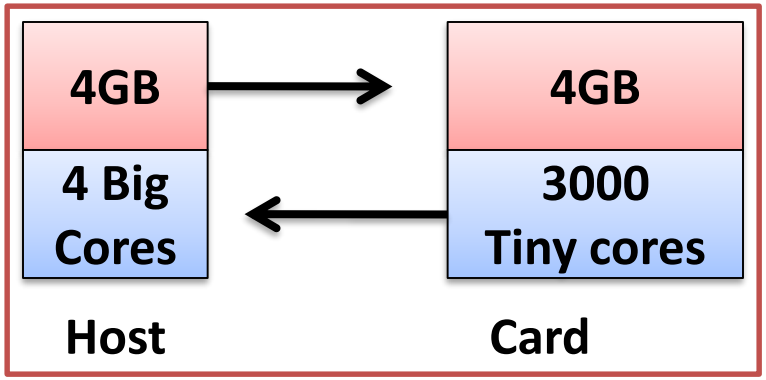
```
int A[100][100], B[100][100],
C[100][100];
for(i=0;i<100;i++)
  for(j=0;j<100;j++){
        Cij=0 ;
        for(k=0;k<100;k++)
          Cij=Cij+A[i][k]*B[k][j];
        }
```



- Executing on Host
  - **$O(n^3)$ time**, zero data transferred
  - $O(n^{2.8})$ Strasen method

Example 3: Matrix Multiplication

```
int A[100][100], B[100][100],
C[100][100];
for(i=0;i<100;i++)
  for(j=0;j<100;j++){
      Cij=0 ;
      for(k=0;k<100;k++)
        Cij=Cij+A[i][k]*B[k][j];
      }
```

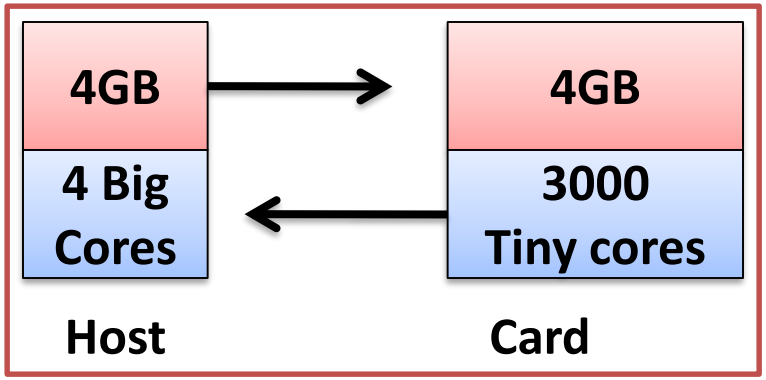


Can be done in one core

- Executing on GPU
  - $2*n^2$ data send from Host to GPU + n^2 GPU to Host
  - **$(n^2/3000)$ * n** time parallel using 3000 cores
  - Significant reduction in running time
  - **This program is a good candidate to run on GPU**

Example 3: Matrix Multiplication

```
int A[100][100], B[100][100],
C[100][100];
for(i=0;i<100;i++)
  for(j=0;j<100;j++){
        Cij=0 ;
        for(k=0;k<100;k++)
          Cij=Cij+A[i][k]*B[k][j];
        }
```



- Executing on the Host
  - Using OpenMP/Pthread
  - Each thread can run for $N^2/4$ Cij computation
  - No need to transfer data, shared among all...
  - **OK, not much reduction in time**

Example 4: N Body Simulation

```
typedef struct pos {float x,y,z;} pos;
typedef Force {float fx,fy,fz} Force;
PosP[N]; float Mass[N]; Force F[N];
while(1){
        for(i=0;i<N;i++){ //Calculate force for all body from others
                F[i]=0;
                for(j=0;j<N;j++)  F[i]=F[i]+force([i][j]);
        }
        for(i=0;i<N;i++){ Ai=Fi/Mi; Pi=f(Ai,Pi);} //update Acc and Pos
        }
}
```

- Executing on Host
  - O(n^2) time for ***each iteration of while loop***..., zero data transferred , **for N>100, Very slow**

Example 4: N Body Simulation

```
while(1){  //for each i, it can be in parallel
        for(i=0;i<N;i++){ //Calculate force for all body from others
                F[i]=0;
                for(j=0;j<N;j++)
                        F[i]=F[i]+G*M[i]*M[j]/r[i][j]*r[i][j];
        } //for each i, it can be in parallel
        for(i=0;i<N;i++){
                A[i]=F[i]/M[i]; P[i]=f(A[i],P[i]);} //update A &P
        }
}
```

- Executing on GPU (**Embarrassingly parallel**)
  - 1*n data send from Host to GPU (old positions), 1*n from GPU to Host, onetime transfer of Mass
  - **Execution in O(N) time parallel using N cores (if N<3000)**
  - Total time: 1*n + O(N) + 1*n = O(n)