# Basics of GPU  Architecture & Programming

- Single core/CPU Performance essentially getting saturated;

- Multi-core CPU based chips – supportive to a large class of application problems until 2011/2012;

- Sudden increase in demand for computational power – Big Data, HPC needs, medical and financial data processing, high-precision engineering;

*Solution ?*

- Parallelization – Code and Data;
- What is the architecture that supports this realization?
- What is the programming model?

# Graphics Processing Unit (GPU)

Graphics processing is "inherently parallel" and there is a lot of parallelism that an be exploited; Typically we can say, O(pixels) is the computational demand;

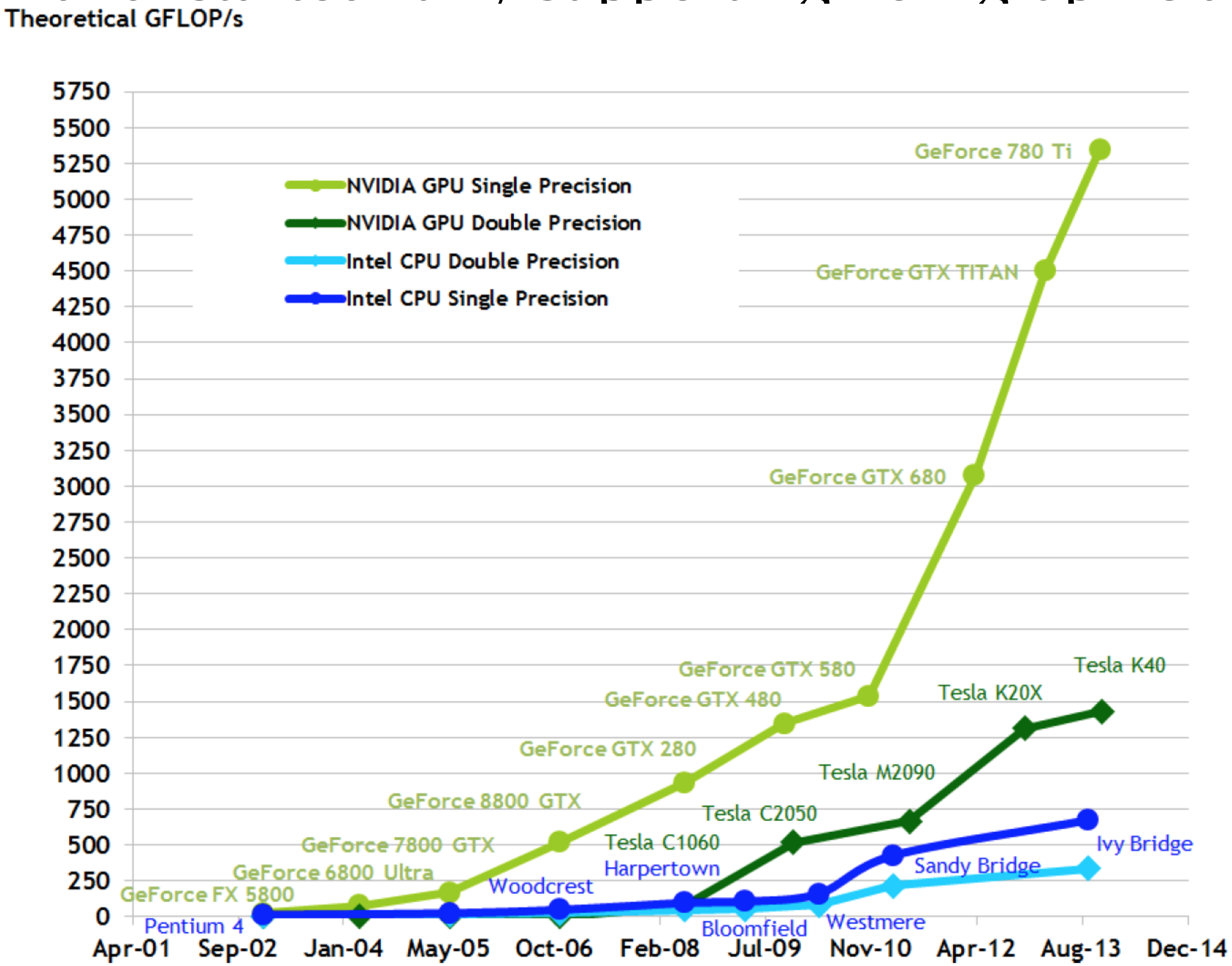
Initial designs targeted two aspects:

•Graphics applications;
•Hardwired (less programmable) to provide speed and parallelism;

~2005 onwards GPUs started getting attention for non-graphic computations and accommodated programming!

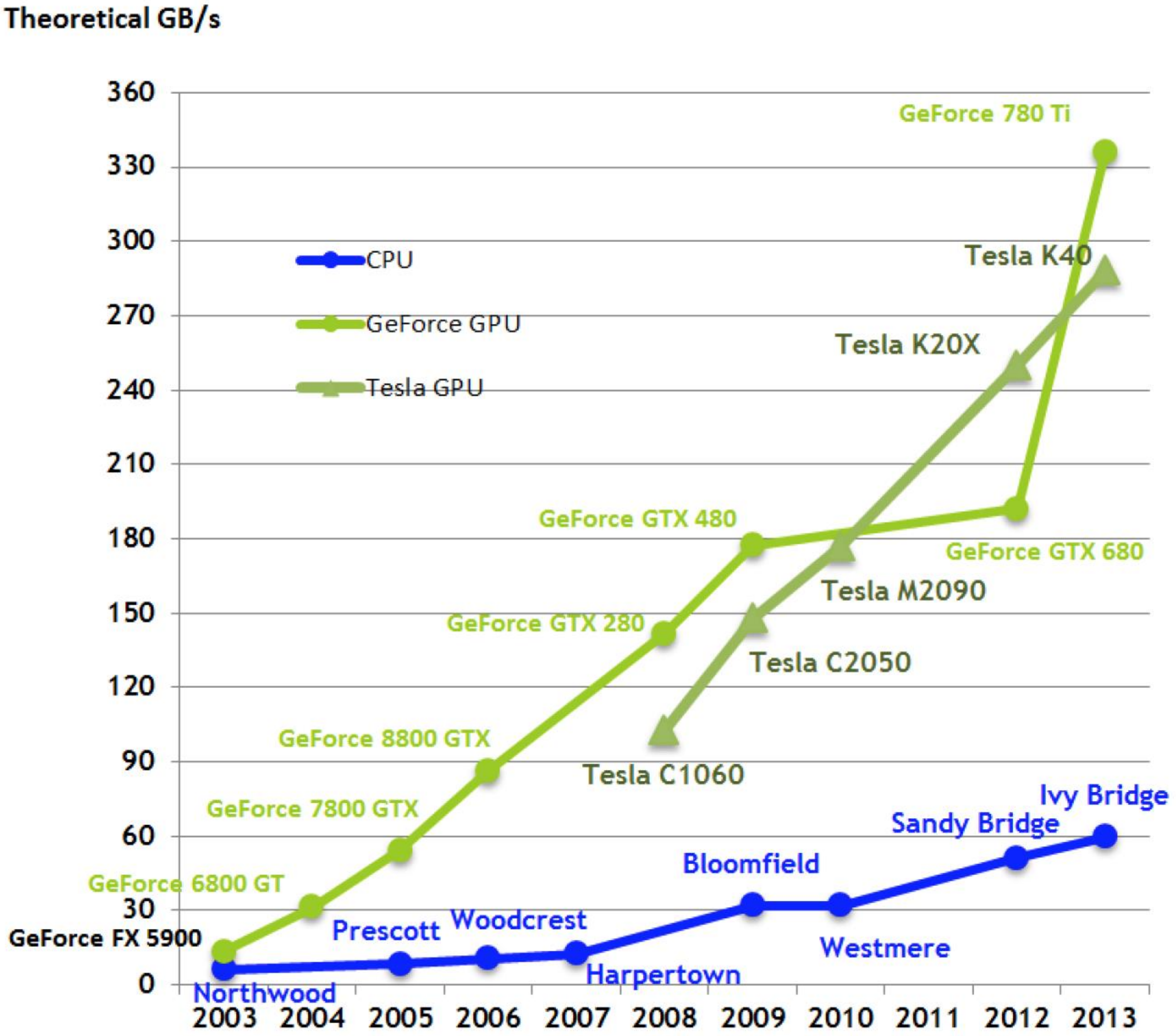
# Throughput performance - GPU

**Year 2006** – Nvidia releases “CUDA” language for GPUs and  started fully supporting non-graphic applications;



Src: NVIDIA CUDA Programming guide

# Memory Bandwidth performance - GPU



# Example:

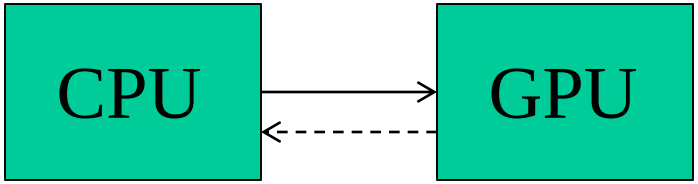
Kepler K40 GPUs from NVIDIA have performance
of 4TFlops (peak)
– CM-5, #1 system in 1993 was ~60 Gflops (Linpack)
– ASCI White (#1 2001) was 4.9 Tflops (Linpack)

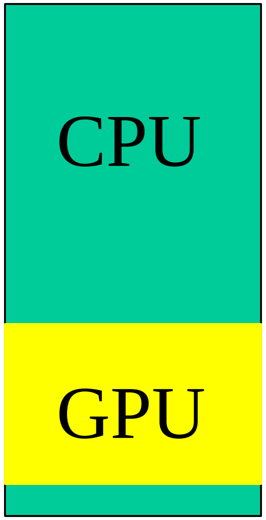
As GPUs became very powerful alternate CPU choices could not stand the competition (IBM Cell, RSX,etc)

Current day GPUs are also referred to as “accelerator-based systems” (*Why?*)

# Three types of GPU

1. CPU+GPU combo; Offers more computational power and memory bandwidth; Discrete type;



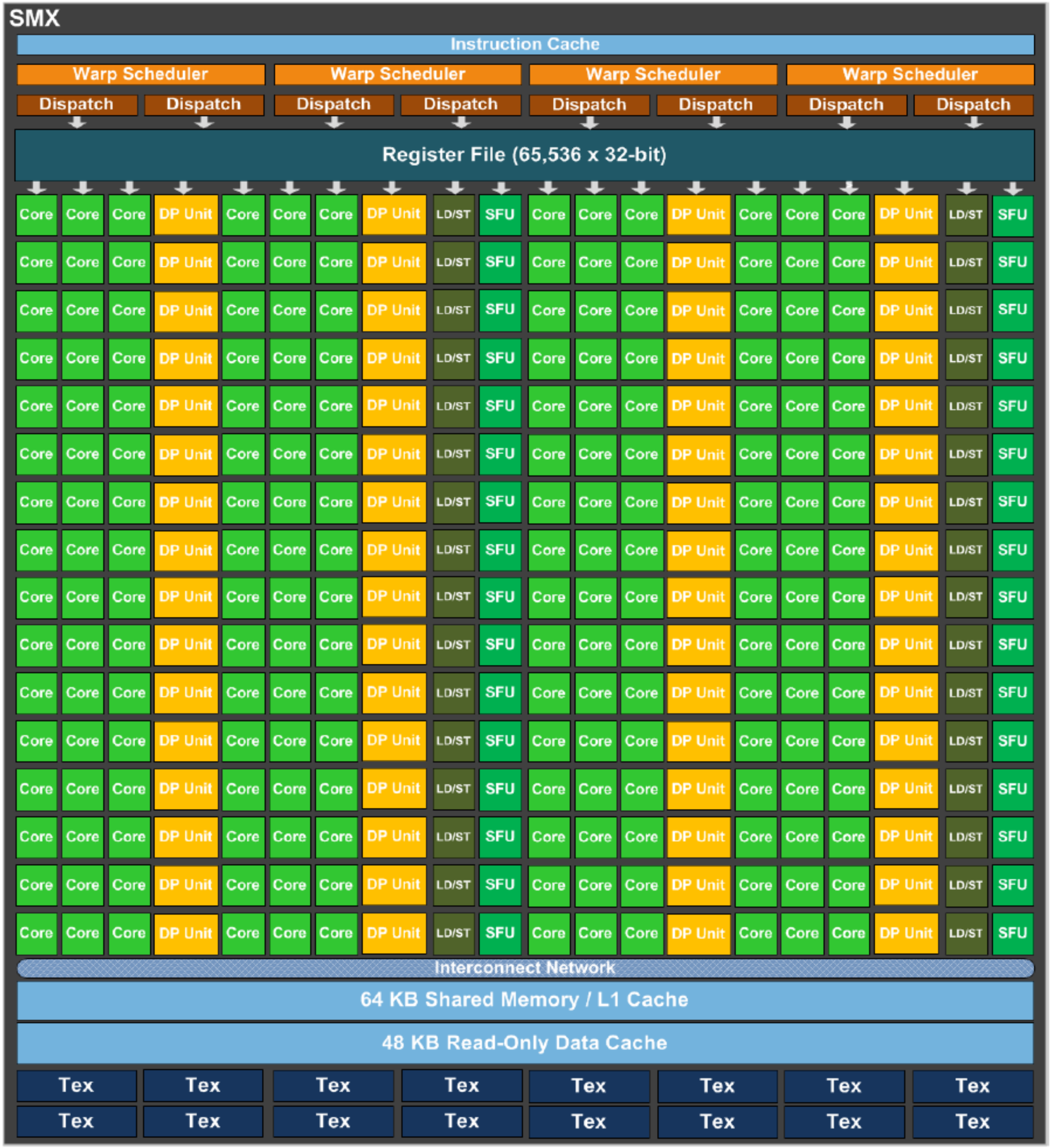


2. Types 2 &3 – Integrated GPUs

Shared-memory System;

Energy considerations

# Nvidia Kepler SMX Architecture



L1 – can be partitioned

Courtesy: Nvidia, Kepler Whitepaper

# How to use a GPU?

Some common golden rules:

1. You must retarget code for the GPU
2. The working set must fit in GPU RAM
3. You must copy data to/from GPU RAM
4. Data accesses should be streaming
5. Use scratchpad as user-managed cache
6. Lots of parallelism preferred (throughput, not latency)
7. SIMD-style parallelism best suited
8. High arithmetic intensity (FLOPs/byte) preferred

# Remarks: Caches vs Scratchpads

CPU caches are "automatically" managed by the hardware together with OS-  when the requested memory contents are not in the cache, fetches that data from main memory.

In modern architectures, caches are generally a hierarchy, with level 1 caches being very fast to access, but small, and level 2 and higher caches being larger and slower.

Most of the time, a programmer hopes that accesses hit in L1 or L2 caches, so the processor spends little or no time waiting on memory.

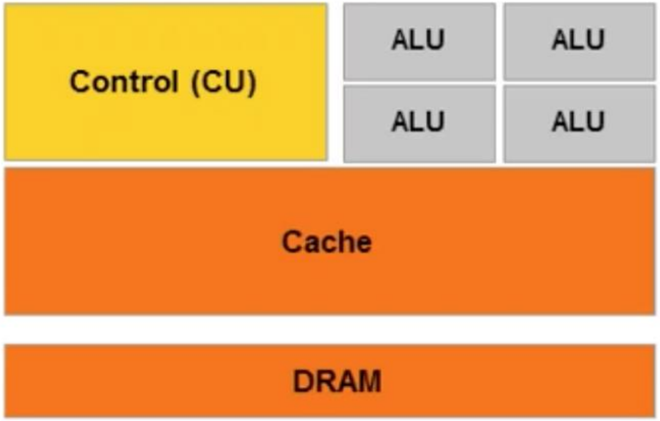
Scratchpad memories are "manually" managed: a program explicitly addresses the memory, write the results and retrieve them.

Scratchpads are usually relatively small, on the order of L1 or L2 caches, fast (1-2 cycle access), and often more importantly, exhibit deterministic behavior/performance;

This means, if you write in a scratchpad, the data are always there and ready to go, unlike a cache, where the contents might have been evicted and need to be retrieved.

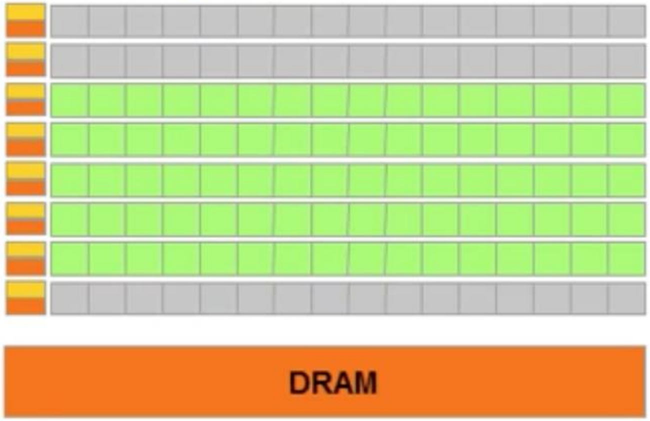
# CPU vs GPU

## Multi Core
## Latency Oriented CPU



- Less no. of core
- Higher Clock Freq.
- Lower Bandwidth
- Less FLOPS/watt
- Less Peak Performance (TFLOPS)
- More total Memory

## Many thread
## Throughput Oriented GPU



- More no. of core
- Lower Clock Freq.
- Higher Bandwidth
- More FLOPS/watt
- More Peak Performance
- Less total Memory
- Heterogeneous: GPU attached to CPU via slow PCIe

Courtesy: Nvidia Kepler Whitepaper

# SIMD/T model for GPUs

Data Parallel Computation

- All cores execute single instruction at any given clk cycle;
- Each instruction operates on different data elements;
- SIMT - Version of SIMD used in GPUs;
- GPUs use a thread model to achieve very high parallel performance and to hide memory latency;
- Multiple threads, each execute the same instruction sequence; On a GPU, a very large number of threads (10K) possible;
- Threads mapped onto a  available processors on GPU (1000's of processors all executing same program sequence)

# Basics of GPU CUDA Programming

CUDA - Compute Unified Device Architecture - Heterogeneous model

- Architecture & programming model - NVIDIA, 2007

- Enables GPUs to execute programs written in C, Fortran, etc

  CUDA C – special APIs, syntax and kernel functions added to C to in the design of CUDA C

- CUDA code has two components – Host & Device;

  Host component (executes on CPU) & Device component (on GPU)

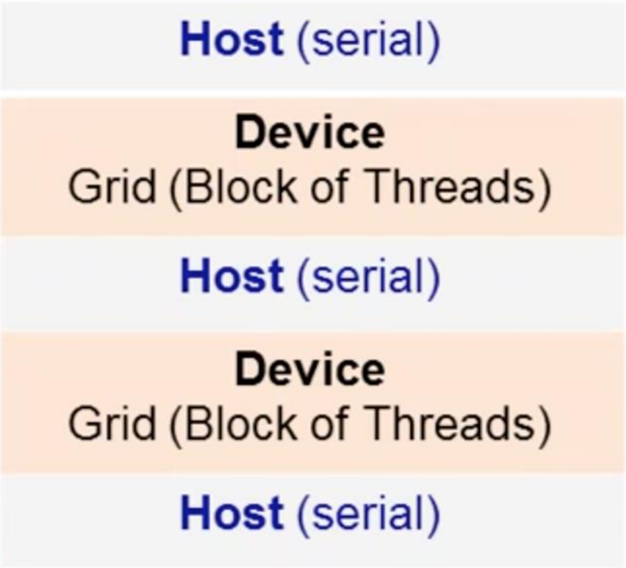
- Within C programs, we can call SIMT "kernel functions" that are executed on GPU; Kernels – data parallel functions that aid execution on GPUs;

# Basics of GPU Programming

Heterogeneous :
CPU → host (own memory) traditional compiler
GPU→ device (own memory) CUDA compiler

Functions of CPU:

1. Execute serial code;
2. Allocate GPU memory;
3. Copy data CPU to GPU;
4. Launch "kernel" on GPU;
5. Copy data GPU to CPU;
6. Error handling;



**Host** (serial)

**Device**
Grid (Block of Threads)

**Host** (serial)

**Device**
Grid (Block of Threads)

**Host** (serial)

Kernels (data parallel portion)→ serial program executed by threads
Possible to launch hundreds/thousands of thread (on cores)

# Basics of GPU Programming

Written in C with extensions - CPU code (runs on CPU) & GPU code (runs on GPU)

CUDA keywords separate the host and device codes;

- CPU allocates storage on GPU - cuda malloc()
- CPU copies input data from CPU to GPU – GPU memory;
- CPU launches kernels on GPU to process data - kernel launch;
- GPU runs lots of threads in parallel
- CPU copies results back to CPU from GPU - cud memcpy();

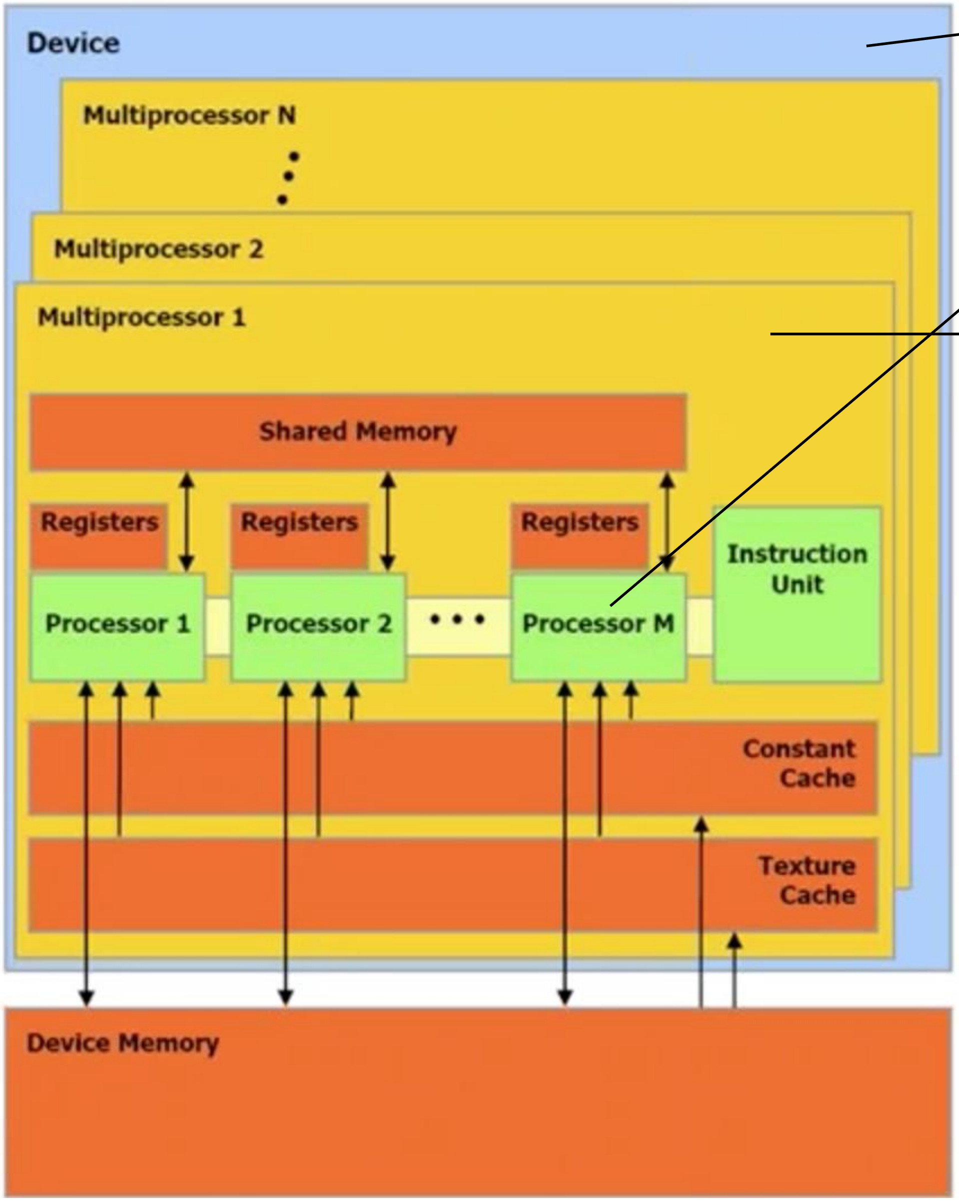
# Basics of GPU Programming

Vector addition example using CUDA C

```
int main() {
 // declare arrays A,B,C and intialize
 for(i=0;i<n;i++)
          C[i]=A[i]+B[i];
}
```

1. Declare CUDA variables
float *d_A,*d_B,*d_C;
2. Allocate GPU memory for A,B,C using cudamalloc(…)
3. Launch the **kernel** <…> to Actually perform vector + using a number of threads;
4. Copy back the results to CPU using cudamemcpy(…)

# Basics of GPU Software Architecture



Array of SMs = GPU

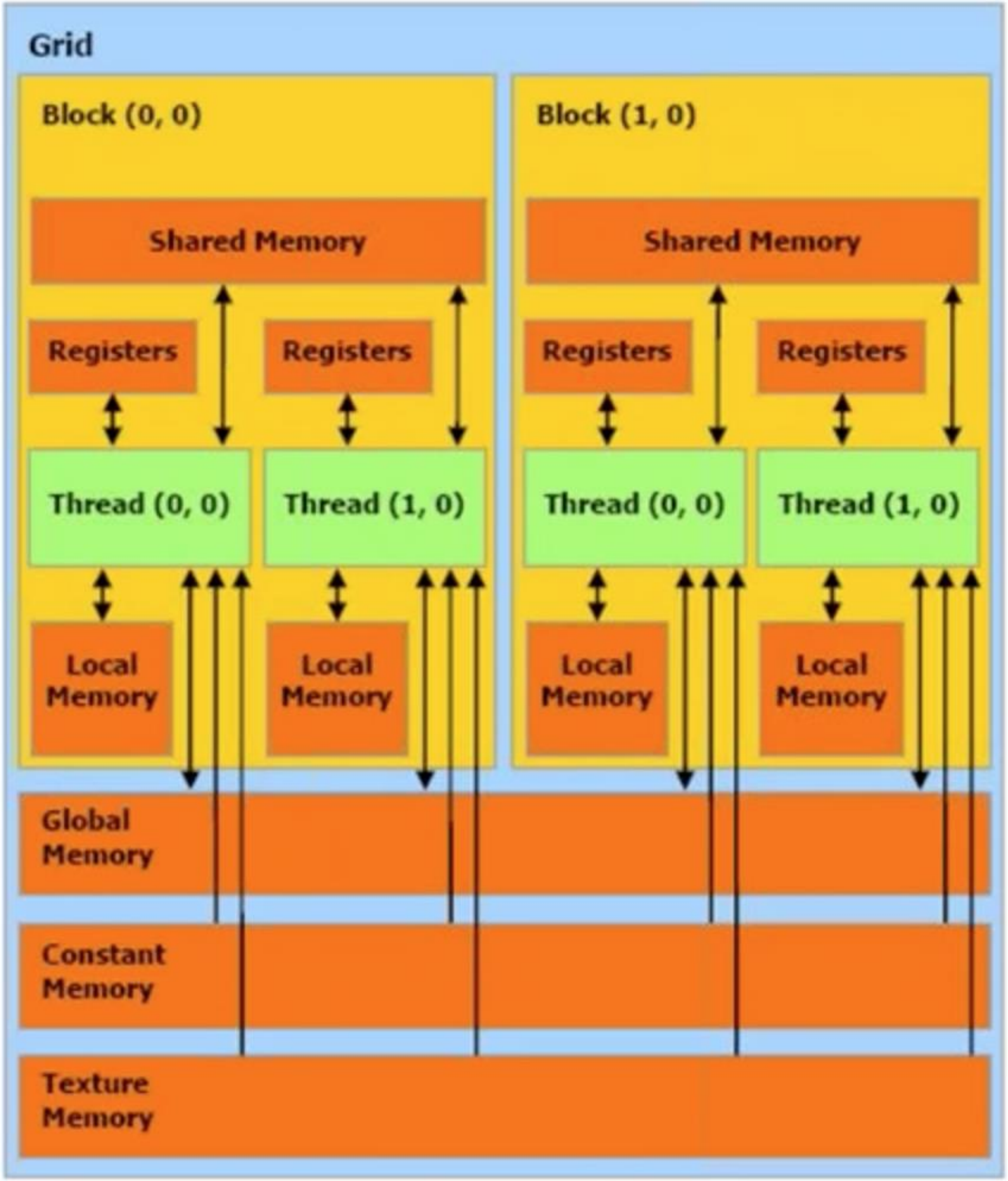
Streaming processor (SP) "core"

Streaming multiprocessor (SM) – comprising several SPs

Memories:

- Shared (local to SM)
- Registers (local to cores)
- Device memory (global memory)

# Basics of GPU Programming



A thread corresponds to a core;

Collection of threads form a block; So, a block corresponds to a multiprocessor;

Collection of blocks forms a Grid; So **grid is the kernel** that is launched on the GPU by the CPU;

Observe “local mem” & “registers” per thread and a shared mem per block;

# Basics of GPU Programming

Local private memory - per thread
Shared memory - per block
Global memory - per application

- Each core can access any memory at ~100s Gb/s, but with different latencies;
- Shared memory - small latency;
- Device memory - ~100x slower than shared;

- GPU executes kernel grids;
- SMs executes one or more thread blocks;
- SM executes threads in groups of 32 threads called a “warp”

# Basics of GPU Programming

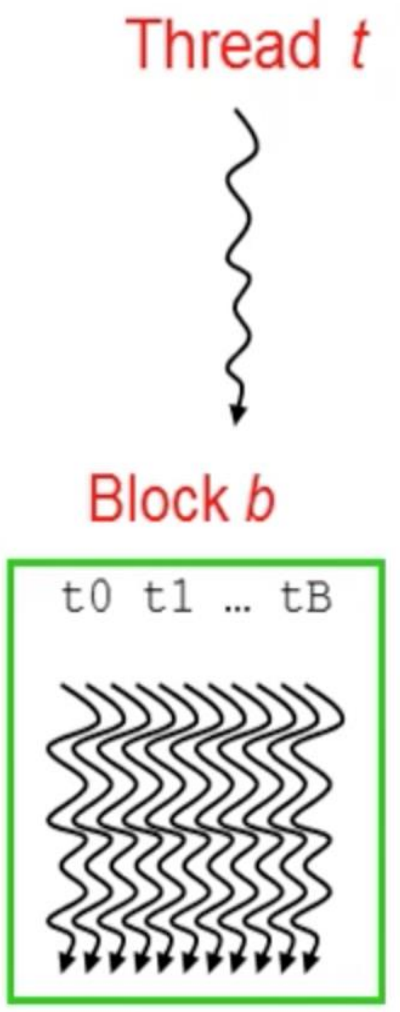
## Thread hierarchy

Kernels composed of many threads;
All threads execute the same code independently;

Threads are grouped into “thread blocks”
Threads in the same block can cooperate;

Blocks are grouped into a Grid
Threads/blocks have unique IDs;

Hardware virtualization mapping:
Thread – Virtualized SP/core;
Block – Virtualized SM



# Basics of GPU Programming

The “kernel” code is a regular C code, however, instead of usual array index, we use “thread IDs” as indices to access data;

```
        index = ThreadID;
        C[index] = A[index]+B[index];
```

*How do we identify codes that are executed on CPU (host) and GPUs(device)?*

| **Functions** | **Executed on the:** | **Only Callable from the:** |
|---|---|---|
| __device__ float DeviceFunc() | DEVICE | DEVICE |
| __global__ void KernelFunc() | DEVICE | HOST |
| __host__ void HostFunc() | HOST | HOST |

# Basics of GPU Programming

**Vector addition example** (Refer to slide #14) – adding two one-D arrays;

1.Allocate memory space in “host” for data

```
// regular C
int *h_a, *h_b, *h_c;
…
h_a = (int*)malloc(size);
h_b = (int*)malloc(size);
h_c = (int*)malloc(size);
```

2. Allocate memory in “device”(GPU) for data

```
// we use CUDA malloc routines

int size = N*sizeof(int); //space for N ints
int *devA, *devB, *devC; // device pointers

cudaMalloc((void**) &devA,size));
cudaMalloc((void**) &devB,size));
cudaMalloc((void**) &devC,size));
```

# Basics of GPU Programming

## 3.Transferring data from CPU to GPU

For this, we use cuda routine called: cudaMemcpy()

cudaMemcpy(devA, h_A, size, cudaMemcpyHostToDevice);

cudaMemcpy(devB, h_B, size, cudaMemcpyHostToDevice);

where, devA & devB points to destination n the device;
h_A & h_B are pointers to Host data; 4th parameter specifies the direction of transfer;

## 4. Declaring Kernel routine to run on GPU

Kernel call from Host code;
This contains information of threads using two parameters:

myKernel <<< **n, m**>>>(arg1,…);
where:
n - number of blocks to be used
m - number of threads in this block

For this example: we will set: n=1, m=N

arg1,… specifies arguments - typically pointers to device memory obtained from cudaMalloc() earlier;

# Basics of GPU Programming

*How do we write this kernel?*

```
 vecAdd<<<1,N>>>(devA,devB,devC);
// Grid has 1 block with N threads in a blk;
```

*How do we actually launch it? (launched from the Host)*

```
__global__ void vecAdd(int *A, int *B, int *C);
{
  int i = threadldx.x; // cuda structure that provides thread id in the block
  C[i] = A[i]+B[i];
}
```

Note:

Thread j:  devC[j] = devA[j]+devB[j], j=0,…,N-1

# Basics of GPU Programming

## 5.Transferring data from GPU to CPU

For this, we use cuda routine called: cudaMemcpy() as before;

cudaMemcpy(C, devC, size, cudaMemcpyDeviceToHost);

where, devC is a pointer in the device; C is the pointer in the host;

4th parameter specifies the direction of transfer;

## 6. Free the memory!

Frees the object from device global memory; has only one parameter - pointer to the object to be freed;

```
cudaFree(devA);
cudaFree(devB);
cudaFree(devC);
```

## *Complete code here!*

```
#define N 256

__global__ void vecAdd(int *A, int *B, int *C) {
   int i = threadldx.x;
   C[i] = A[i]+B[i];
}

int main(int *argc, char **argv[]) {  /* run on Host */
                int size = N*sizeof(int);
                int a[N],b[N],c[N],*devA,*devB,*devC;
                cudaMalloc((void**) &devA,size);
                cudaMalloc((void**) &devB,size);
                cudaMalloc((void**) &devC,size);
                cudaMemcpy(devA, h_A, size, cudaMemcpyHostToDevice);
                cudaMemcpy(devB, h_B, size, cudaMemcpyHostToDevice);
                vecAdd<<<1,N>>>(devA,devB,devC);
                cudaMemcpy(C, devC, size, cudaMemcpyDeviceToHost);
                cudaFree(devA);
                cudaFree(devB);
                cudaFree(devC);
                return(0);
}
```

# Basics of GPU Programming

*Compilation?*

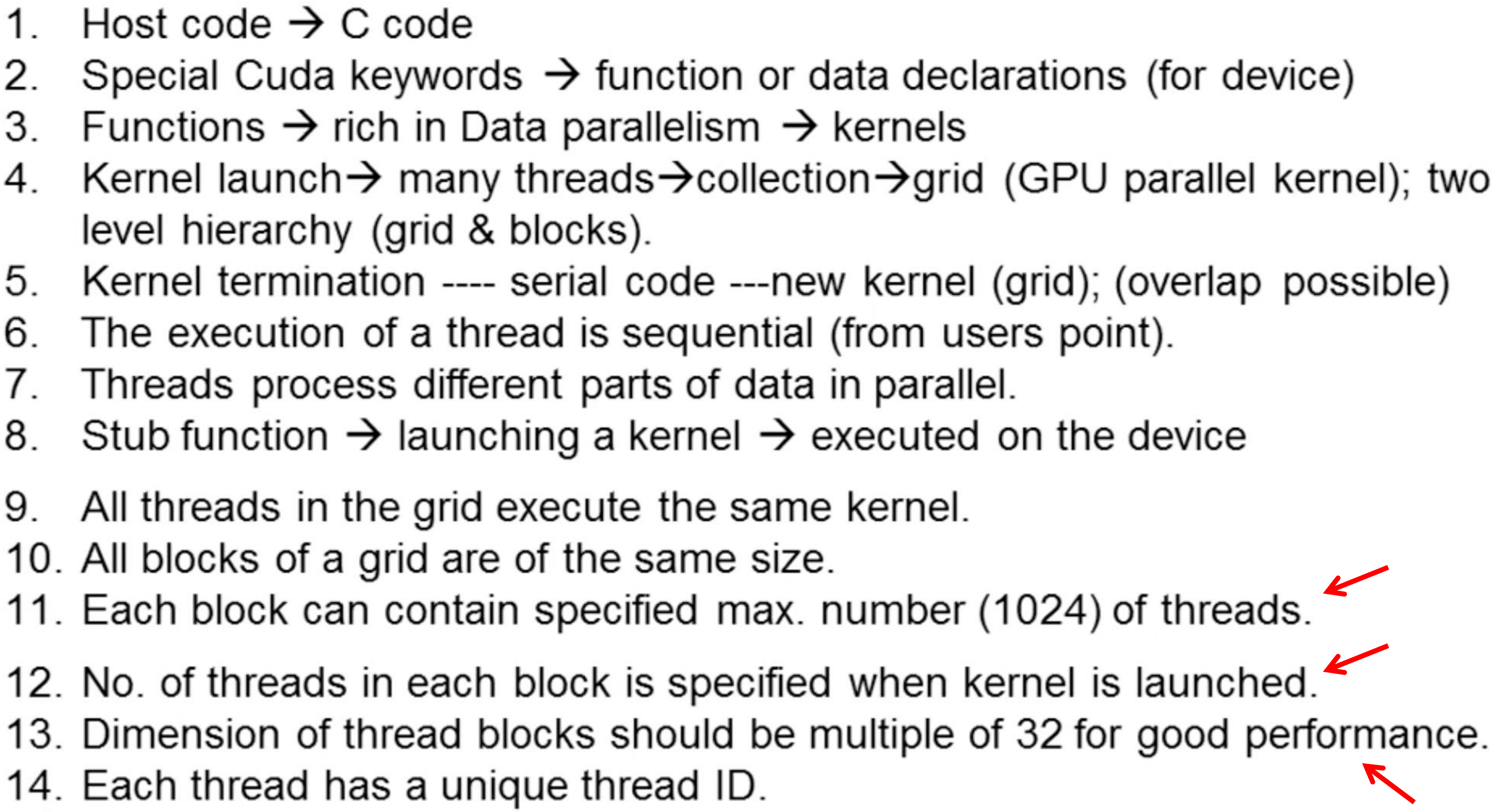
Download NVIDIA toolkit and it has the required compiler;

NVIDIA provides “nvcc” – compiler + driver

nvcc will automatically separate out the code for
Host and Device; It uses:

gcc – for normal C/C++ on host;
nvcc – for the device code;

# Summary



1. Host code → C code
2. Special Cuda keywords → function or data declarations (for device)
3. Functions → rich in Data parallelism → kernels
4. Kernel launch→ many threads→collection→grid (GPU parallel kernel); two level hierarchy (grid & blocks).
5. Kernel termination ---- serial code ---new kernel (grid); (overlap possible)
6. The execution of a thread is sequential (from users point).
7. Threads process different parts of data in parallel.
8. Stub function → launching a kernel → executed on the device
9. All threads in the grid execute the same kernel.
10. All blocks of a grid are of the same size.
11. Each block can contain specified max. number (1024) of threads.
12. No. of threads in each block is specified when kernel is launched.
13. Dimension of thread blocks should be multiple of 32 for good performance.
14. Each thread has a unique thread ID.

# Concluding remarks

CUDA performance optimization depends on a number of factors;

We have two main time-consuming aspects:

Computation & Memory access

Major issues:

- synchronization via __synchTheads();
- Overheads (communications, computations)
- Strategies to hide latency;
- Thread divergence issues;
- Memory access efficiency;