# 2DV605

# Parallel Computing

## GPU Programming

Suejb Memeti and Sabri Pllana

Department of Computer Science

Building D, Room D2236C, LNU, Växjö

https://sabripllana.eu/

# Outline

- ❑ **Introduction**

- ❑ **CUDA**

- ❑ **Example: Matrix Multiplication**

# Computing Platforms

Credit: Intel

- ❑ **Multi-core – seeks to maintain the speed of sequential programs while moving into multiple cores**
  - Consists of few cores optimized for sequential processing
  - Known as - **Host**
    - ex: Intel Multi-core CPUs

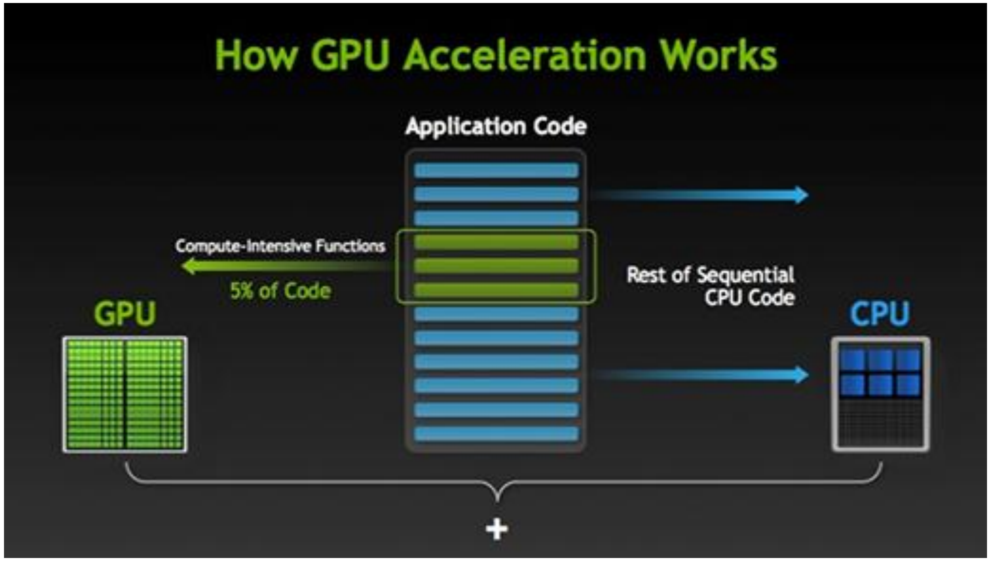
- ❑ **Many-core – focuses on the execution throughput of parallel applications**
  - Consists of thousands of smaller cores optimized for parallel execution
  - Peak floating point performance is about 10x faster compared to the performance of multi-core CPUs.
  - Known as - **Device**
    - ex: NVIDIA GPU, Intel Xeon Phi, FPGA

Credit: NVIDIA

- ❑ **This performance gap has motivated developers to move the computationally intensive parts of their applications to the GPUs**

# Heterogeneous / Accelerated Computing

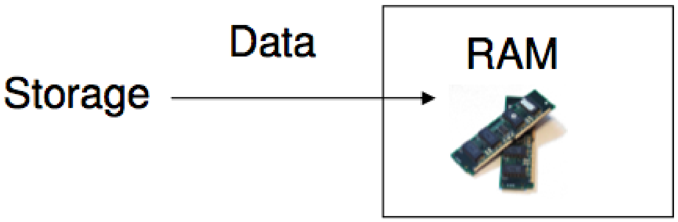
- ❑ **Parts of the code (usually compute intensive tasks) are executed on the GPU**
  - Known as **Offloading**
- ❑ **The remainder of the code still runs on the CPU**


How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
GPU
Rest of Sequential CPU Code
CPU
+

Credit: NVIDIA

# Homogeneous Computing

- ❑ **Load data to the RAM**

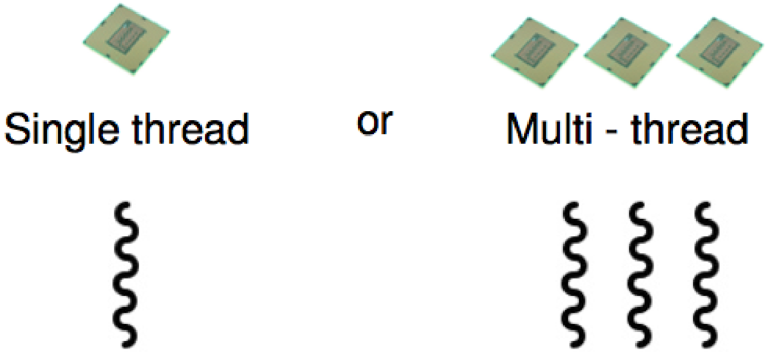


Data

Storage

RAM

- ❑ **Process data with CPU computation cores**

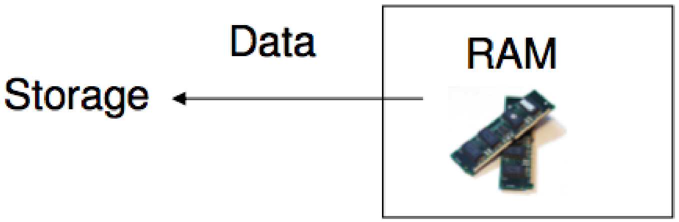


Single thread

or

Multi - thread

- ❑ **Store result**



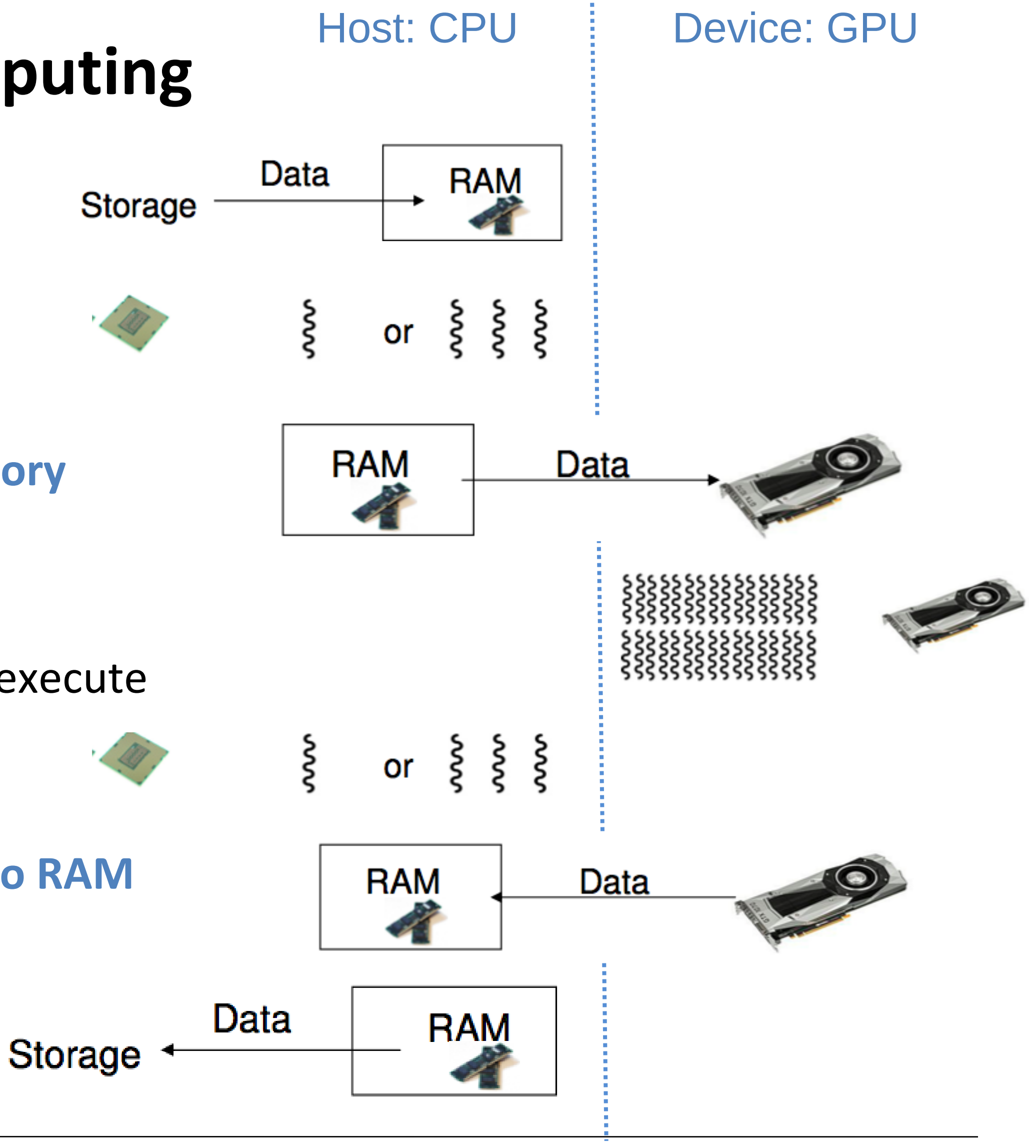
Data

Storage

RAM

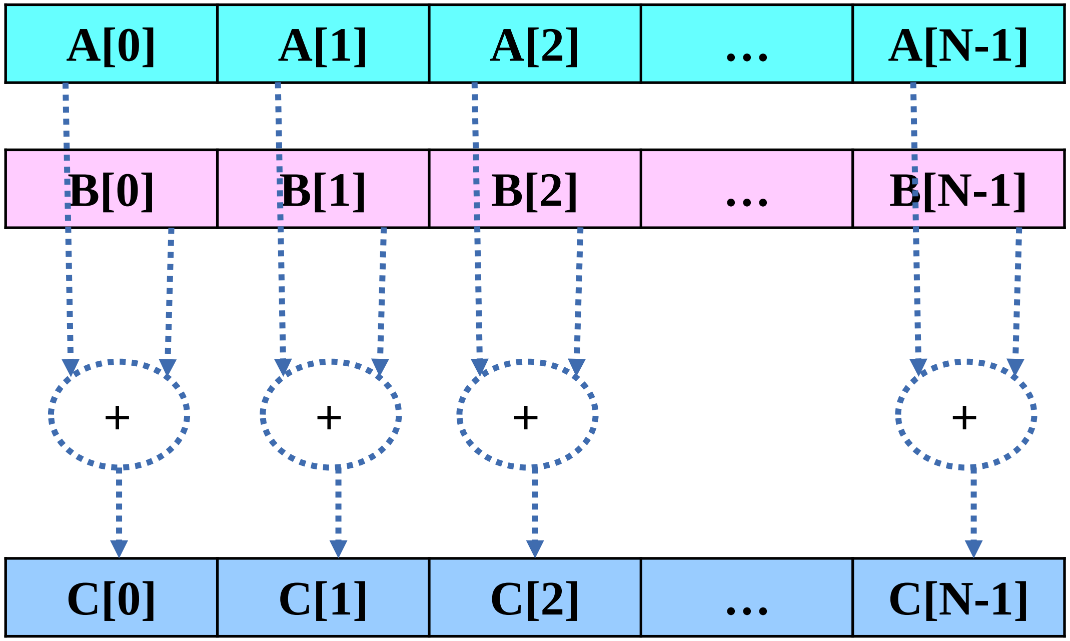
# Heterogeneous Computing

- □ **Load data to the RAM**
- □ **Pre-process on CPU**
- □ **Load data to the GPU memory**
- □ **Process data with GPU**
  - CPU can either wait or execute some job
- □ **Copy data back from GPU to RAM**
- □ **Store result**



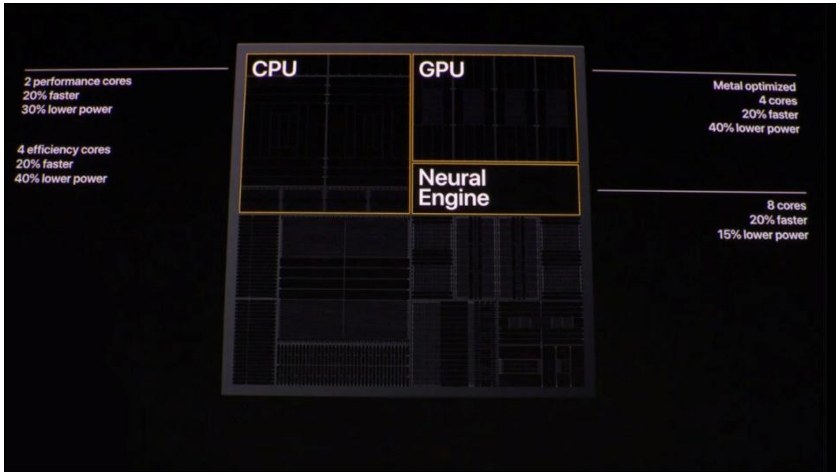
# Data Parallelism – Vector Addition Example

- ❑ Vector A
- ❑ Vector B
- ❑ Vector C



# Graphics Processing Unit (GPU)

- ❑ **GPUs are everywhere**
  - smart-phones
  - supercomputers
- ❑ **Traditionally used for graphics rendering**
  - Computer games
  - Other 3D rendering applications
- ❑ **General Purpose GPUs**
  - Can be used to accelerate various scientific and engineering applications



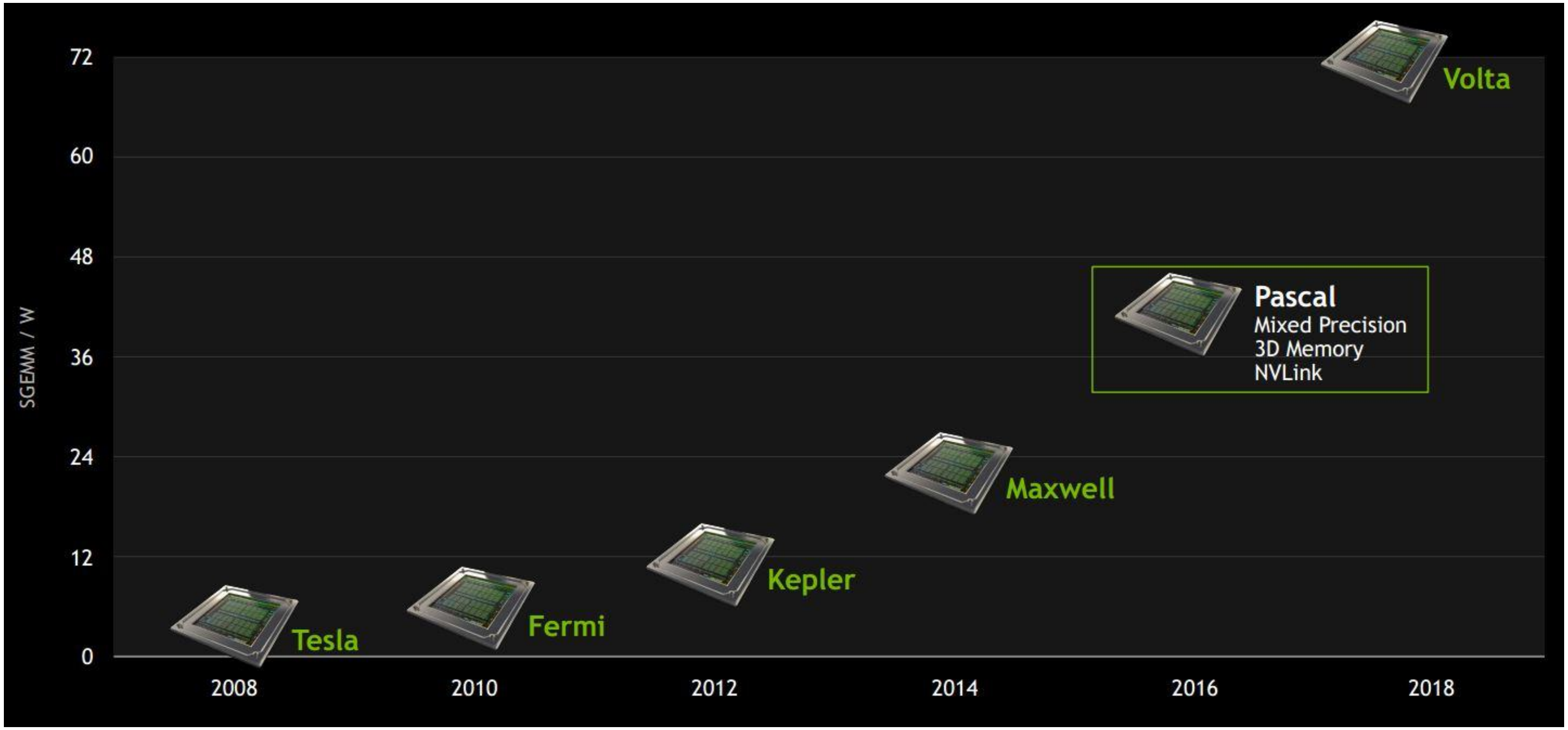
Apple A13 Bionic processor is used in iPhone 11

# Supercomputer Example: Summit at ORNL

- *Rank 1 in TOP500 list (June 2019)*
- *200 petaflops (max performance)*
- *250 petabytes of data storage*  
  *- 74 years of HD video*
- *4608 computing nodes*
- *each node comprises*  
  *- two IBM Power9 22-core CPUs*  
  *- six NVIDIA Volta GPUs*
- *340 tons*
- *space of two tennis courts*

Credit: NVIDIA & ORNL

# GPU Roadmap



Single precision floating General Matrix Multiply (SGEMM), 16K x 16K SGEMM

Credit: NVIDIA

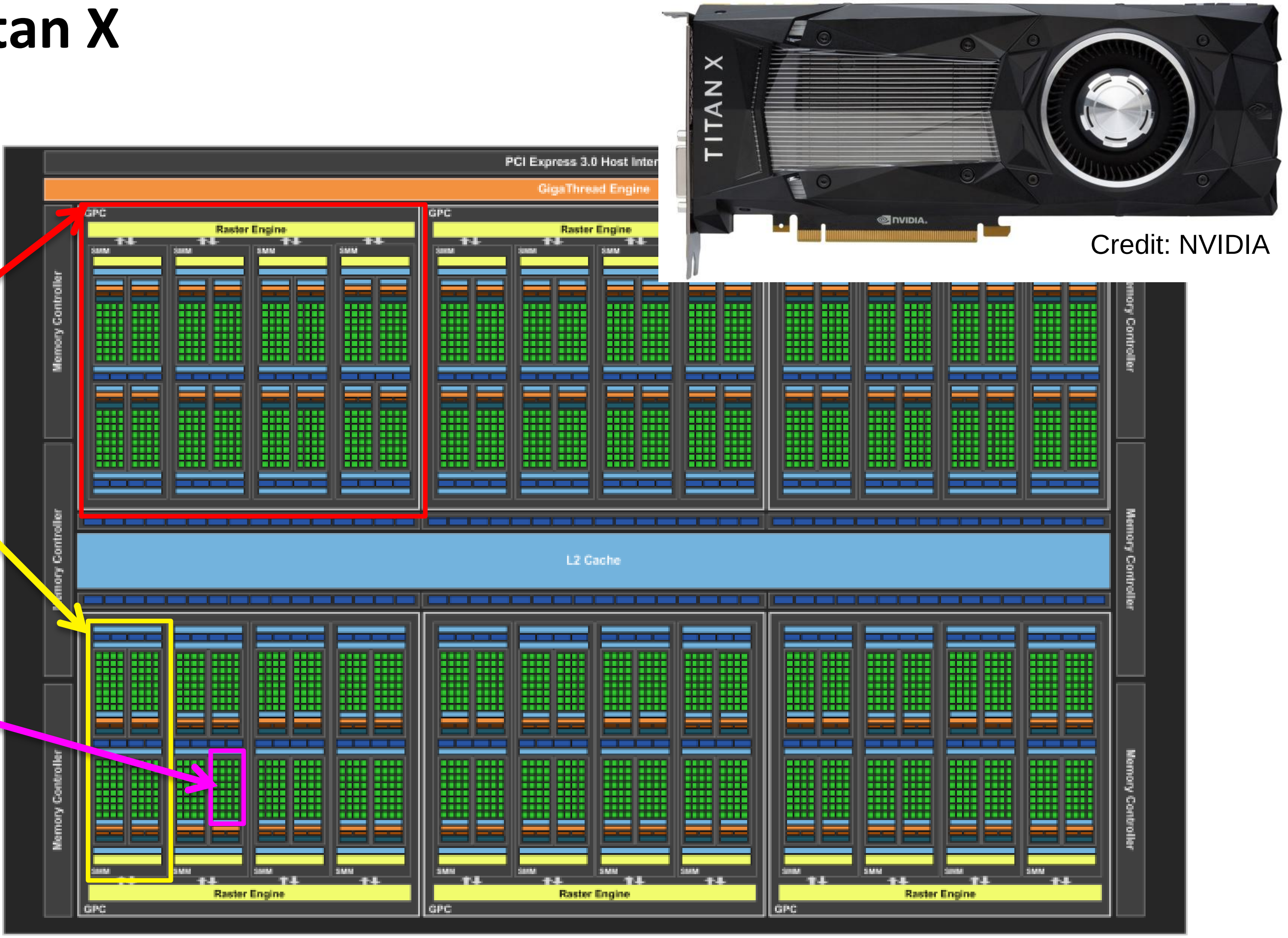
# Ida @ LNU

- ❑ **Host**
  - 2 x Intel Xeon E5-2650 v4 CPUs
  - 2.2– 2.9 GHz
  - **12 Cores per CPU**
  - 24 Threads per CPU
  - 30MB cache
  - Max. Memory Bandwidth is 76.8GB/s
  - 384GB Memory
  - 105W TDP

- ❑ **Device**
  - 1 x GeForce GTX Titan X GPU
  - 1 – 1.1 GHz
  - 24 SMs
  - **3072 cores**
  - 336.5 GB/s Memory Bandwidth
  - 12 GB Memory
  - 250W TDP
  - Warp Size is 32
  - Maximum threads per block 1024

# GTX Titan X

- □ GP cluster
- □ Streaming Multiprocessor
- □ Cores



Credit: NVIDIA

# CUDA

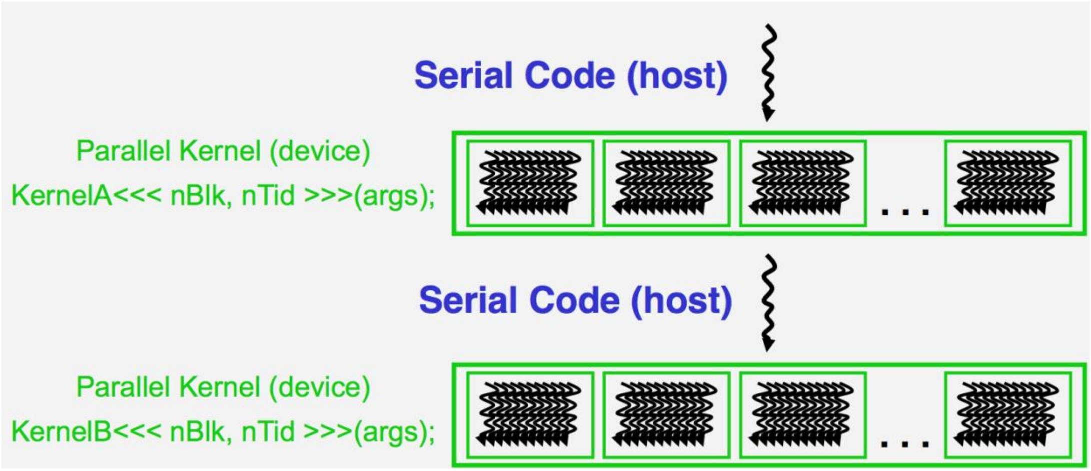
# CUDA

- ❑ **A heterogeneous parallel programming interface that enables exploitation of data parallelism**
  - Hierarchical thread organization
  - Main interfaces for launching parallel execution
  - Mapping of thread index to data index

- ❑ **CUDA version installed on Ida @ LNU**
  - 8.0

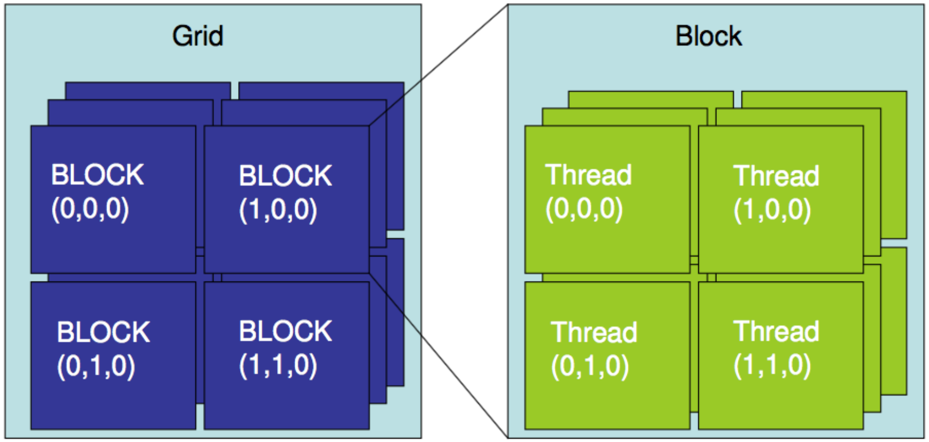
# Execution Model

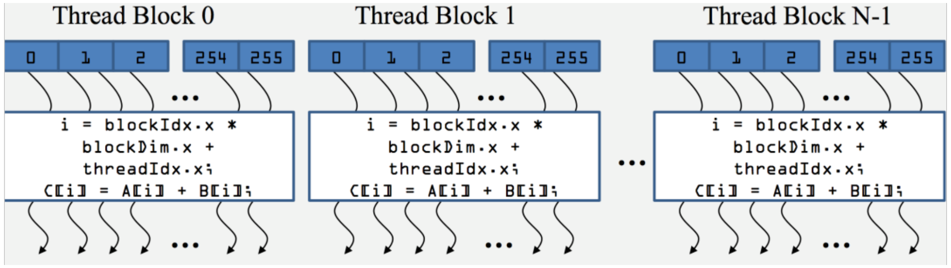
- ❑ **Heterogeneous (host + device) C application**
  - Serial parts of the code are executed on the host
  - Parallel parts of the code (CUDA kernels) are executed on the device



# Grid, Block, Thread

- ❑ **Grid – 3D Matrix of blocks**
- ❑ **Block – 3D Matrix of threads**
- ❑ **Thread – computation unit**
  - Threads within a block communicate via the shared memory, atomic operations, and barrier synchronization
  - Threads in different blocks do not interact



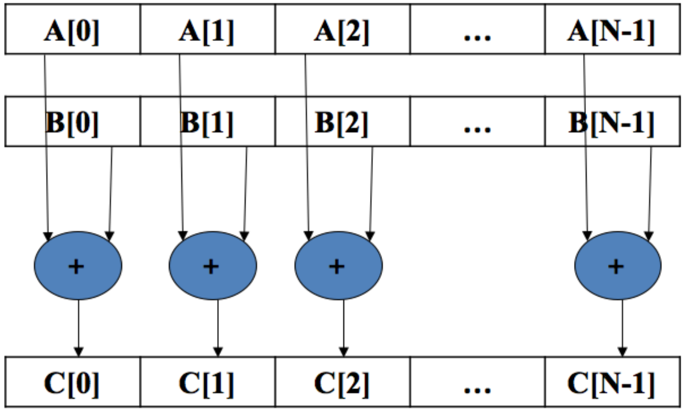


```
i = blockIdx.x *
blockDim.x +
threadIdx.x;
C[i] = A[i] + B[i];
```

# Computation Kernels

- ❑ **A CUDA kernel is executed by an array of threads (so called grid)**
  - **All threads within a grid run the same kernel code**
  - Each thread has indexes, which are used to compute the memory addresses that enable to access the data items

- ❑ **The vector addition example**
  - We determine the index using the built-in variables:
    - gridDim, blockIdx, blockDim, threadIdx, warpSize
  - Ex:
    - *i = blockIdx.x * blockDim.x + threadIdx.x;*
      *C[i] = A[i] + B[i];*



# Grid, Block, Thread

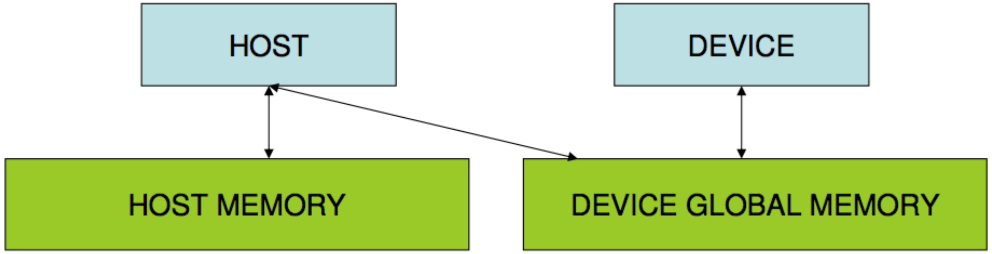
❑ **Any call to a CUDA kernel must specify the execution configuration**

- grid size
- block size
- number of threads per block blockDim
- number of blocks per grid gridDim

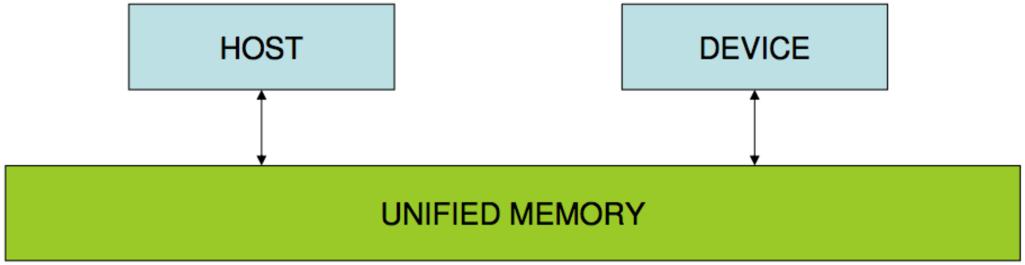
```
__global__ void vecAdd(double *a, double *b, double *c, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i < n) // make sure we do not go out of bounds
    c[i] = a[i] + b[i];
}
// executing the kernel
blockSize = 1024;
gridSize = (int)ceil((float)n/blockSize);
VecAdd<<<gridSize, blockSize>>>(a, b, c, n);
```

# Memory Model

- **NUMA**
  - Non-unified memory model



- **Unified Memory Model**
  - From CUDA 6.0



# Memory Management

- **cudaMalloc – allocates memory on the device**

- **cudaMemcpy – copies data between host and device**
  - cudaMemcpyKind – CUDA memory copy types

|  |  |
|---|---|
| • cudaMemcpyHostToHost = 0 | Host –> Host |
| • cudaMemcpyHostToDevice = 1 | Host –> Device |
| • cudaMemcpyDeviceToHost = 2 | Device –> Host |
| • cudaMemcpyDeviceToDevice = 3 | Device –> Device |
| • cudaMemcpyDefault = 4 | Default based unified virtual address space |

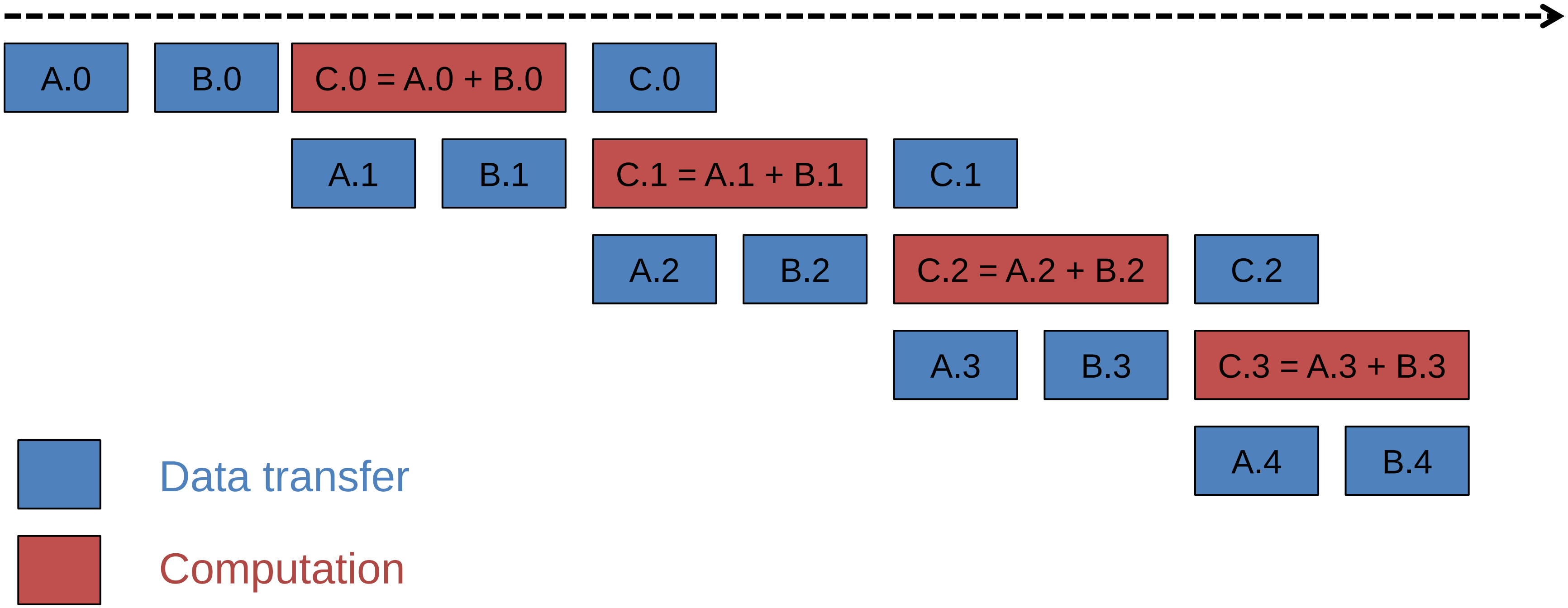
- **cudaFree – frees the memory on the device**

# Basic Memory Management Example

```
…
float *x, *y, *device_x, *device_y; // declare variables
…
x = (float*)malloc(N*sizeof(float)); // allocate memory on host for x
y = (float*)malloc(N*sizeof(float)); // allocate memory on host for y
…
cudaMalloc(&device_x, N*sizeof(float)); // allocate memory on device for x
cudaMalloc(&device_y, N*sizeof(float)); // allocate memory on device for y
…
cudaMemcpy(device_x, x, N*sizeof(float), cudaMemcpyHostToDevice); // copy x from host to device
cudaMemcpy(device_y, y, N*sizeof(float), cudaMemcpyHostToDevice); // copy y from host to device
… // run the computation and compute the result
…
cudaMemcpy(y,device_y, N*sizeof(float), cudaMemcpyDeviceToHost); // copy y from device to host
… // store or print the result
cudaFree(device_x); // free the memory for x on device
cudaFree(device_y); // free the memory for y on device
```

# Asynchronous Transfer

- ❑ **Overlap computation and communication**
- ❑ **Data domain decomposition**
  - Divide large arrays into segments



# Example: Matrix Multiplication

# Environment Variables

- ❑ **NVIDIA CUDA Compiler (NVCC) 8.0 is installed in** **/usr/local/cuda-8.0/bin/nvcc/**

  - ▪ Set the following environment variables
    - PATH:

      **export PATH=$PATH:/usr/local/cuda-8.0/bin:$PATH**
    - LIBRARY_PATH:

      **LIBRARY_PATH=/usr/local/cuda-8.0/lib:/usr/local/cuda-8.0/lib64/:$LIBRARY_PATH**
    - LD_LIBRARY_PATH:

      **export LD_LIBRARY_PATH=/usr/local/lib:/usr/lib:/usr/local/lib64:/usr/lib64:/usr/local/cuda-8.0/lib:$LD_LIBRARY_PATH**

  - ▪ You can put them in **/home/username/.profile** file
    - To automatically load on user login

# Example of .profile File

```
export PATH=$PATH:/usr/local/cuda-8.0/bin:$PATH

export LIBRARY_PATH=/usr/local/cuda-8.0/lib:/usr/local/cuda-8.0/lib64/:$LIBRARY_PATH

export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64/:$LD_LIBRARY_PATH
```

# Initialization

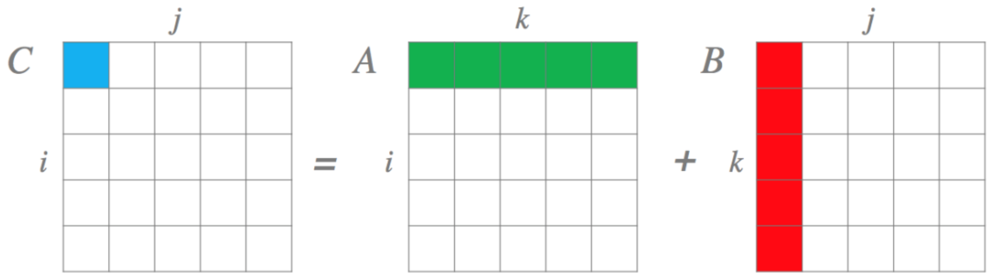
```
#define TILE_SIZE 16
#define MATRIX_SIZE 512

float * hostA; // The A matrix
float * hostB; // The B matrix
float * hostC; // The output C matrix
float * deviceA;
float * deviceB;
float * deviceC;

hostA = (float*) malloc(sizeof(float) * n * n);
…
for(int i = 0; i < n; ++i) {
  hostA[i] = rand()/float(RAND_MAX)*24.f+1.f;
  hostB[i] = rand()/float(RAND_MAX)*24.f+1.f;
  hostC[i] = 0;
}
```

# Data Transfer To / From Device

```
// allocate device memory
cudaError_t err1 = cudaMalloc((void **) &deviceA, sizeA);
cudaError_t err2 = cudaMalloc((void **) &deviceB, sizeB);
cudaError_t err3 = cudaMalloc((void **) &deviceC, sizeC);

// Copy memory to the GPU
cudaMemcpy(deviceA, hostA, sizeA, cudaMemcpyHostToDevice);
cudaMemcpy(deviceB, hostB, sizeB, cudaMemcpyHostToDevice);
cudaMemcpy(deviceC, hostC, sizeC, cudaMemcpyHostToDevice);

// execute the kernel

// Copy the GPU memory back to the CPU
cudaMemcpy(hostC, deviceC, sizeC, cudaMemcpyDeviceToHost);

// Free the GPU memory
cudaFree(deviceA);
cudaFree(deviceB);
cudaFree(deviceC);
```

# Matrix Multiplication on Host

```
void mmultHost(float * A, float * B, float * C, int numARows,
int numACols, int numBRows, int numBCols, int numCRows, int
numCCols) {
  for (int i=0; i < numARows; i ++) {
    for (int j = 0; j < numACols; j++) {
      for (int k = 0; k < numCCols; k++) {
        C[i*numCCols+j] += A[i*numACols+k] * B[k*numBCols+j];
      }
    }
  }
}
```



# Matrix Multiplication on Device

```
dimBlock(TILE_SIZE, TILE_SIZE, 1);
dim3 dimGrid((numCCs/TILE_SIZE) + 1, (numCRows/TILE_SIZE) + 1, 1);
mmultDevice<<<dimGrid, dimBlock>>>(deviceA, deviceB, deviceC, numARows,
numACols, numBRows, numBCols, numCRows, numCCols);
```

```
__global__ void mmultDevice(float *A, float *B, float *C, int numARows,
int numACols, int numBRows, int numBCs, int numCRows, int numCCols) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if(numACols != numBRows) { return; }
  if((row < numARows) && (col < numBCols)) {
    float Cvalue = 0;
    for(int i = 0; i < numACs; ++i) {
      Cvalue += A[row * numACols + i] * B[i * numBCols + col];
    }
    C[row * numCColumns + col] = Cvalue;
  }
}
```

# Compile, Execute, Monitor

❑ **Compile**

- nvcc file_name.cu -o file_name.o

❑ **Execute**

- ./file_name.o {parameters}

❑ **Monitor**

- nvidia-smi
- Watch –n .1 nvidia-smi

# Literature

- Programming Massively Parallel Processors, Third Edition: A Hands-on Approach, 2017 by David B. Kirk and Wen-mei W. Hwu
- Heterogeneous System Architecture: A New Compute Platform Infrastructure 1st Edition by Wen-mei W. Hwu
- Principles of Parallel Programming by Calvin Linand Lawrence Snyder, Pearson Addison Wesley
- CUDA Application Design and Development by Rob Farber
- CUDA by Example: An Introduction to General-Purpose GPU Programming, by Jason Sanders and Edward Kandrot.
- Patterns for Parallel Programming by Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill, Addison Wesley
- CUDA Toolkit reference manual http://docs.nvidia.com/cuda
- CUDA C Programming guide http://docs.nvidia.com/cuda/cuda-c-programming-guide