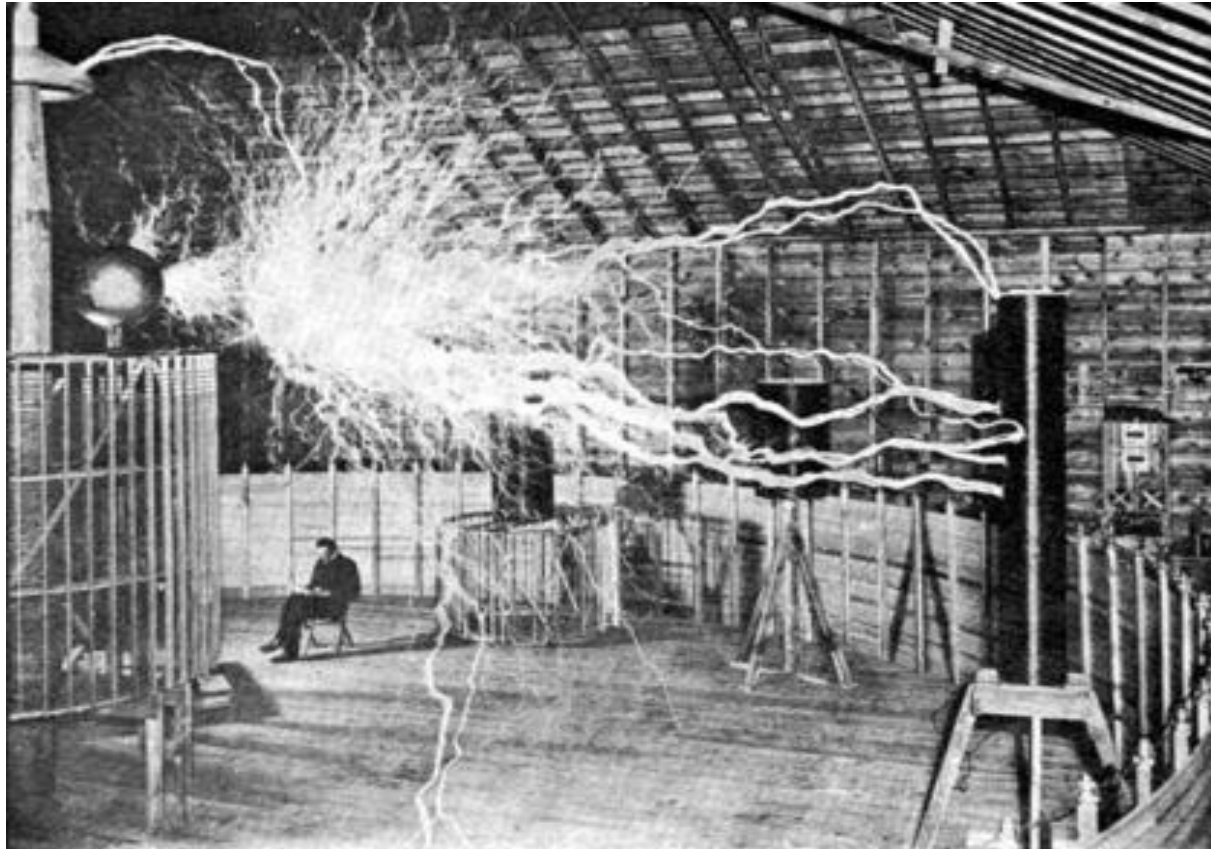


# "Secret Lab: GPU programming for HEP"

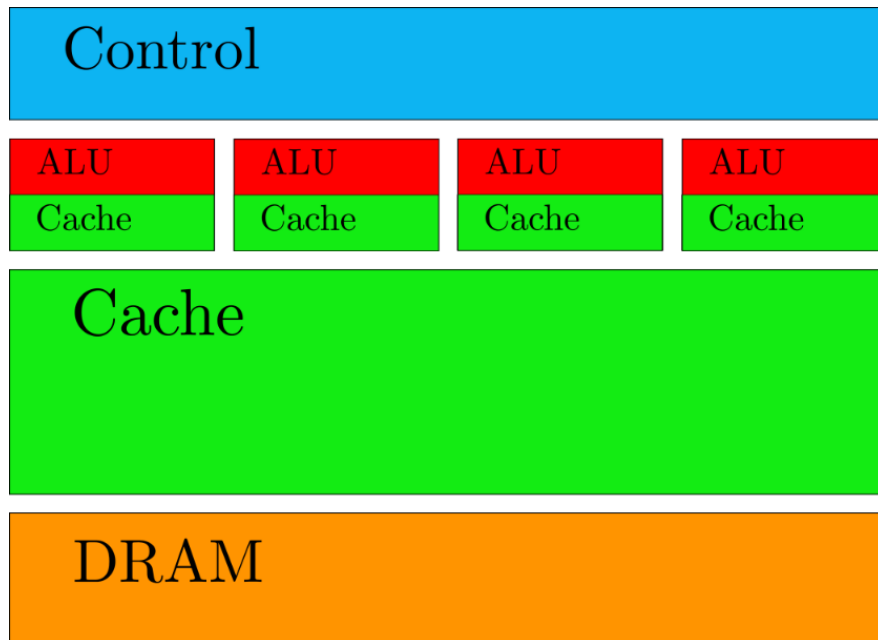


Gianluca Lamanna (Univ. of Pisa & INFN)

## Prerequisites

---

- A little bit of C/C++
- You don't need experience in parallel programming, GPU and computer graphics.

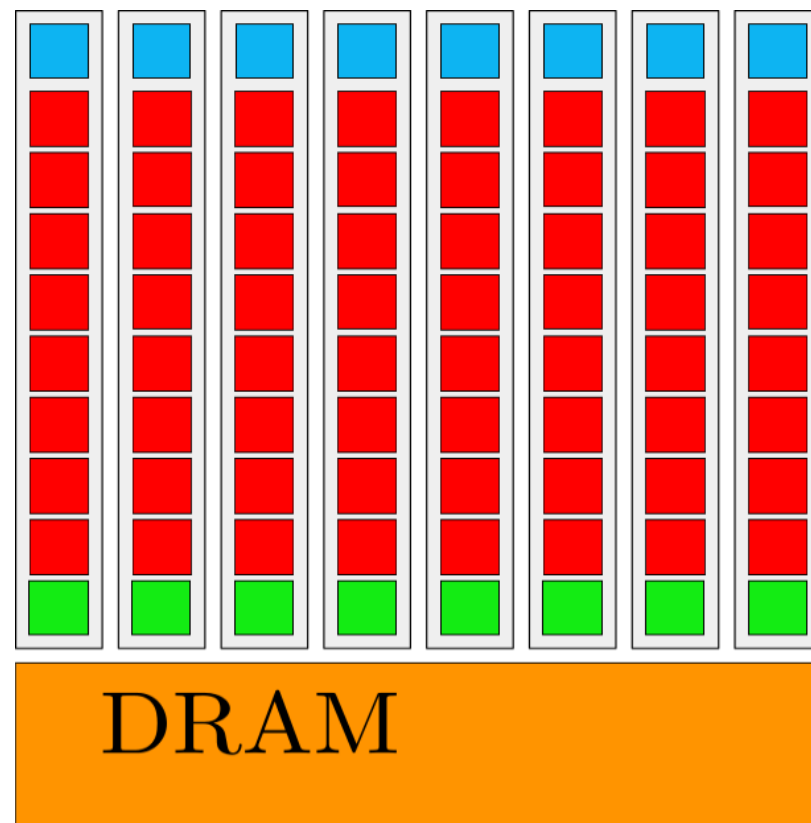


**CPU: latency oriented design**

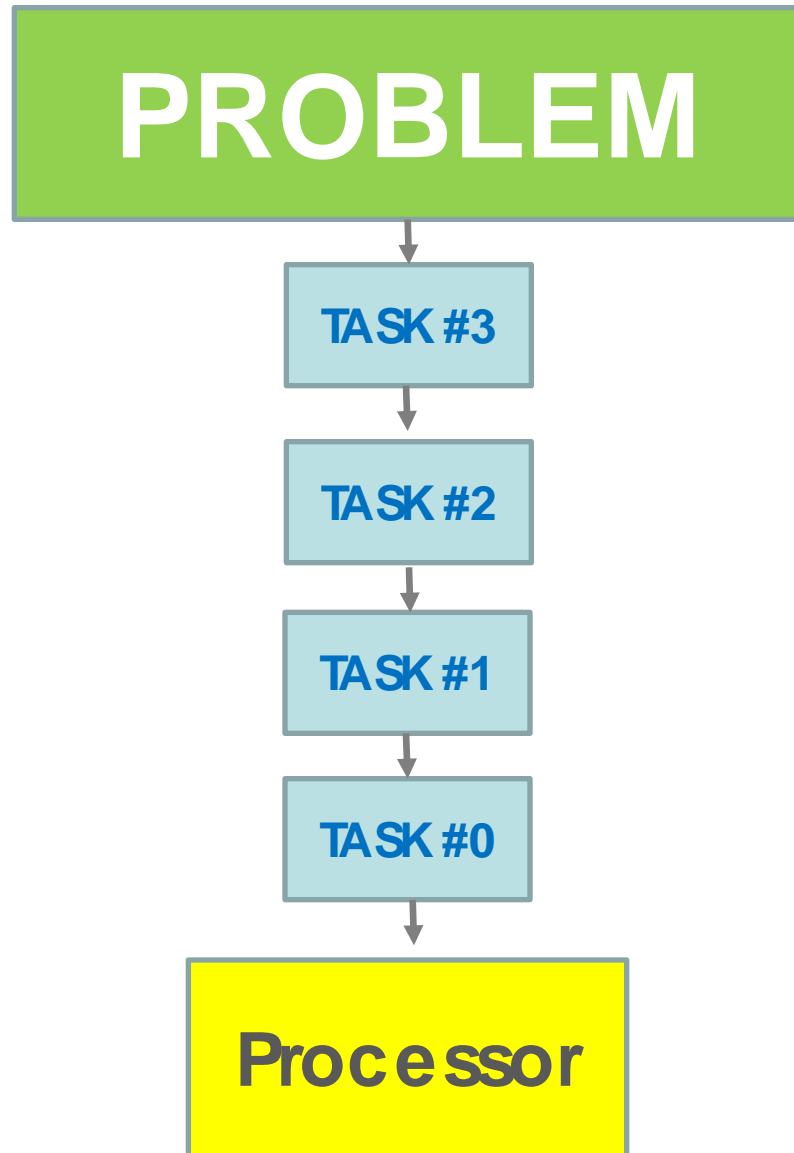
- Multilevel and Large Caches
  - Convert long latency memory access
- Branch prediction
  - To reduce latency in branching
- Powerful ALU
- Memory management
- Large control part

# GPU

- SIMT (Single instruction Multiple Thread) architecture
- SMX (Streaming Multi Processors) to execute kernels
- Thread level parallelism
- Limited caching
- Limited control
- No branch prediction, but branch predication



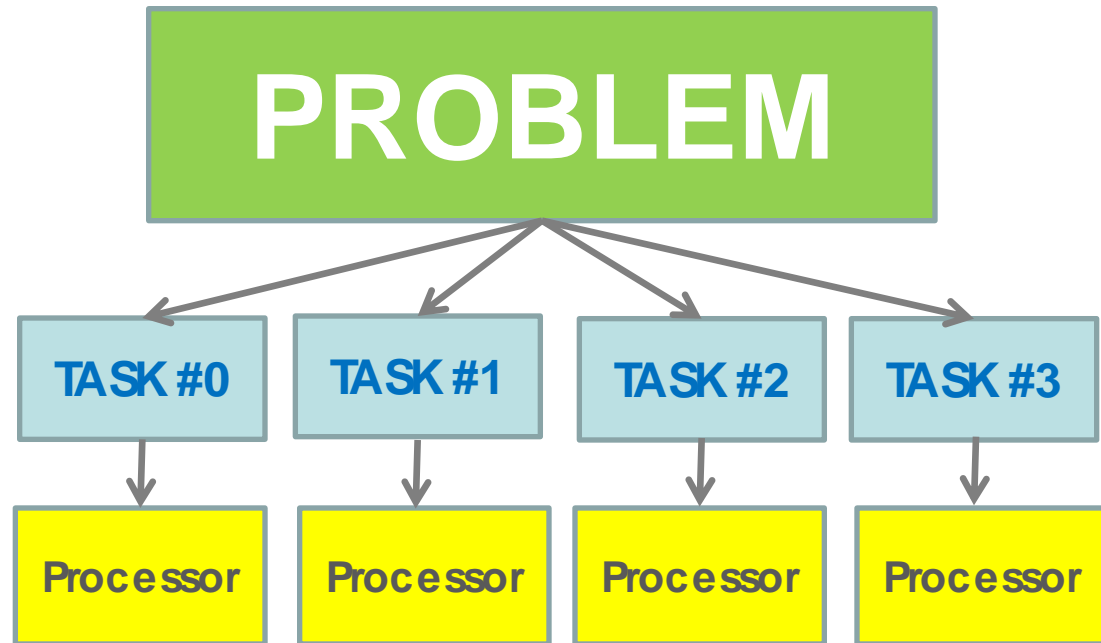
**GPU: throughput oriented design**



- The problem is subdivided in sequential tasks (instructions)
- Only one instruction in each moment
- Often task N depends on the result of task N-1

# Parallel programming

- The tasks are independent at algorithmic level.
- Each task is processed independently on different processors



# Several types of parallelism

---

- Single Instruction Multiple Data (**SIMD**)
  - One scheduler for multiple cores.
  - Different cores execute the same instruction at the same time on different data stream,
- Multiple Instruction Single Data (**MISD**)
  - Different processors execute different instructions on the same data stream
- Multiple Instruction Multiple Data (**MIMD**)
  - Each processor execute its own instruction on its own data set.
- Single Instruction Multiple Threads (**SIMT**)
  - SIMD combined with multithreading

# Several types of parallelism

---

- Single Instruction Multiple Data (**SIMD**)
  - One scheduler for multiple cores.
  - Different cores execute the same instruction at the same time on different data stream,
- Multiple Instruction Single Data (**MISD**)
  - Different processors execute different instructions on the same data stream
- Multiple Instruction Multiple Data (**MIMD**)
  - Each processor execute its own instruction on its own data set.
- Single Instruction Multiple Threads (**SIMT**)
  - SIMD combined with multithreading

**GPUs are SIMT processors!**

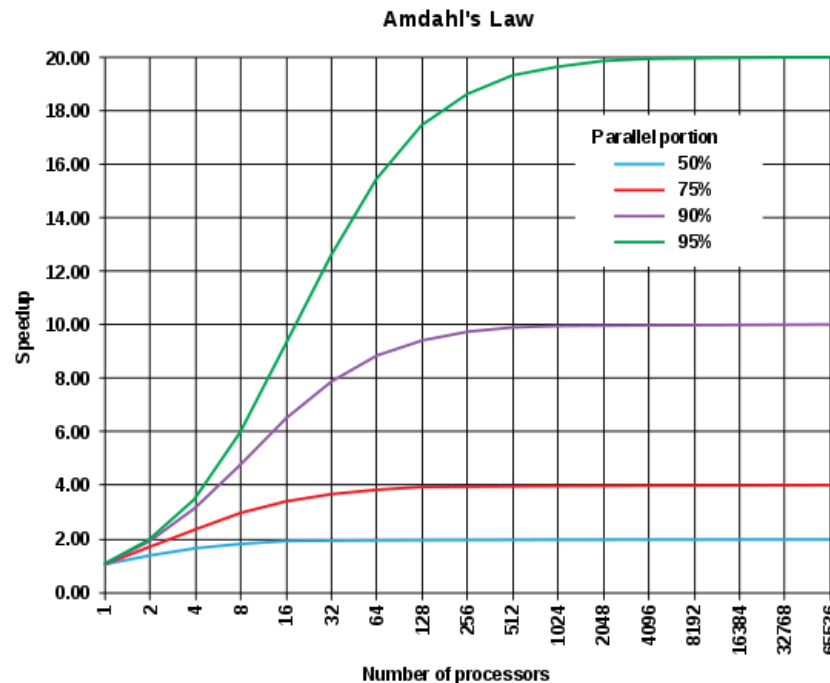
---



# Is it always convenient to use parallel programming?

- Depends on the serial part with respect to the parallel part → **Amdahl's law**:

$$S(p, f) = \frac{T_S}{fT_S + \frac{(1-f)T_P}{p}} \xrightarrow{p \rightarrow \infty} \frac{1}{f}$$



# Is it always convenient to use parallel programming?

- Depends on the serial part with respect to the parallel part → **Amdhal's law**:

$$S(p, f) = \frac{T_S}{fT_S + \frac{(1-f)T_P}{p}} \xrightarrow{p \rightarrow \infty} \frac{1}{f}$$

- The situation improves by increasing the dimension of the problem → **Gustafson's law**

$$S(n) = f(n) + p[1 - f(n)] \xrightarrow{n \rightarrow \infty} p$$

- A winning application use both CPU and GPU
  - CPU for sequential part, where latency matters (CPU can be faster at least x10 than GPU for sequential code)
  - GPU for parallels part where throughput wins (GPU can be faster at least x100 than CPU for parallel core)

# Our system: GTX750

---



- GTX750: for gamers
- Kepler architecture
- 512 cores
- 4 SM (Streaming Multiprocessors)
- 1 Tflops in single precision
- 2 GB Ram
- 80 Gb/s bandwidth
- PCIe 3.0 x16

# TESLA P100

Centernex



- TESLA: for computing
- Pascal architecture
- 3584 cores
- 60 SM (Streaming Multiprocessors)
- 9 Tflops in single precision
- 12-16 GB Ram
- 549 Gb/s bandwidth (or 732 Gb/s)
- PCIe 3.0 x16 (or NVLINK)

# TESLA V100

---



- TESLA: for computing
- Volta architecture
- 5120 cores
- 84 SM (Streaming Multiprocessors)
- 15 Tflops in single precision
- 16 GB Ram
- 900 Gb/s bandwidth
- PCIe 3.0 x16 (or NVLINK)

# How to program GPU?

---

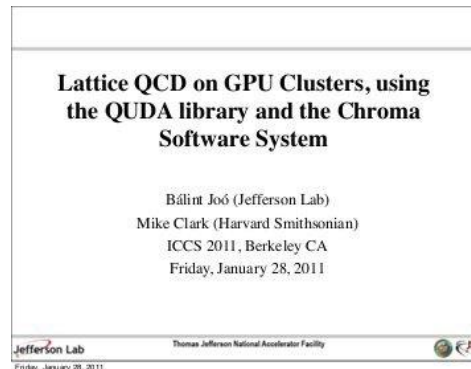
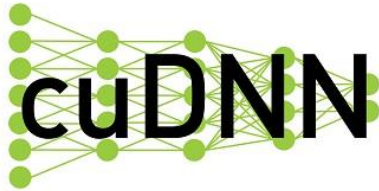
Applications

Libraries

Compiler  
Directives

Programming  
Languages

# GPU Accelerated Libraries



- Easy to use, High Quality
- “plug & play”
- Several library for several applications
- Several examples in physics



# Thrust: Example

---

```
thrust::device_vector<float> deviceInput1(inputLength);  
thrust::device_vector<float> deviceInput2(inputLength);  
thrust::device_vector<float> deviceOutput(inputLength);  
  
thrust::copy(hostInput1, hostInput1 + inputLength,  
             deviceInput1.begin());  
thrust::copy(hostInput2, hostInput2 + inputLength,  
             deviceInput2.begin());  
  
thrust::transform(deviceInput1.begin(),  
                 deviceInput1.end(),          deviceInput2.begin(),  
                 deviceOutput.begin(),  
                 thrust::plus<float>());
```



## OpenACC

Directives for Accelerators

## OpenMP

Enabling HPC since 1997

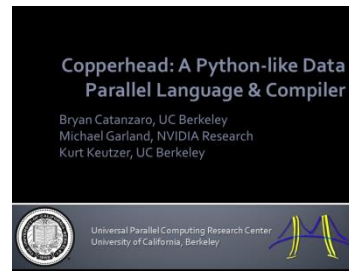
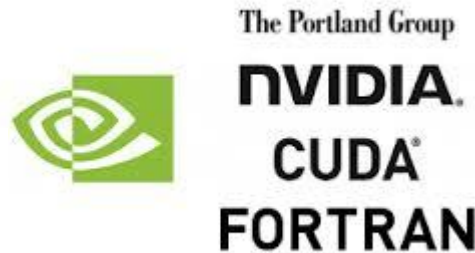
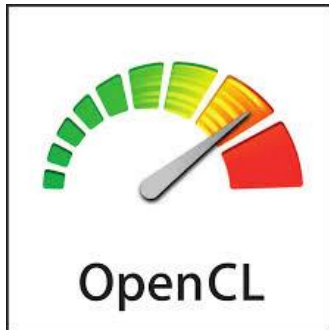
**C++ AMP**  
Accelerated Massive Parallelism  
with Microsoft Visual C++

## OPEN HMPP

- Define a programming model to program accelerators without the complexity associated with the GPU programming
- Easy to use
- Very easy to integrate in already done serial code
- Hardware independent: portable code

```
#pragma acc parallel loop
copyin(input1[0:inputLength], input2[0:
inputLength]),
        copyout(output[0:inputLength])
for(i = 0; i < inputLength; ++i) {
    output[i] = input1[i] +
input2[i];
}
```

# GPU Programming Languages



- Performance: programmers control every computing step
- Flexible: The computation does not need to fit into a limited set of library patterns or directive types
- Complex: the complexity is higher with respect to libraries and directives.

- NVIDIA provides:
  - NVCC: compiler for device+host, device, host applications
  - CUDA MEMCHECK: debugger for memory
  - CUDA-GDB: parallel debugger
  - Nvvp, nvprof: Profilers
  - Nsight: An IDE platform
  - Tons of examples

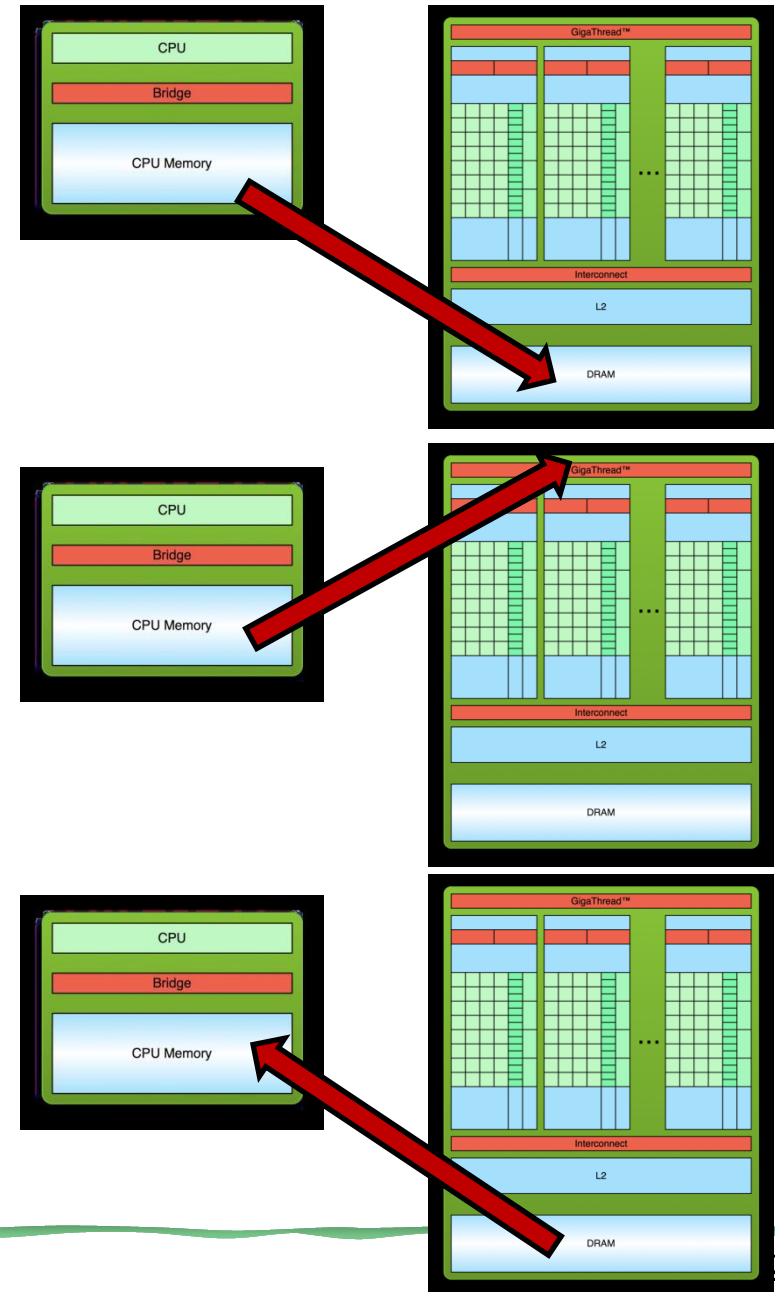
<https://developer.nvidia.com/cuda-downloads>

<https://docs.nvidia.com/cuda/>

---

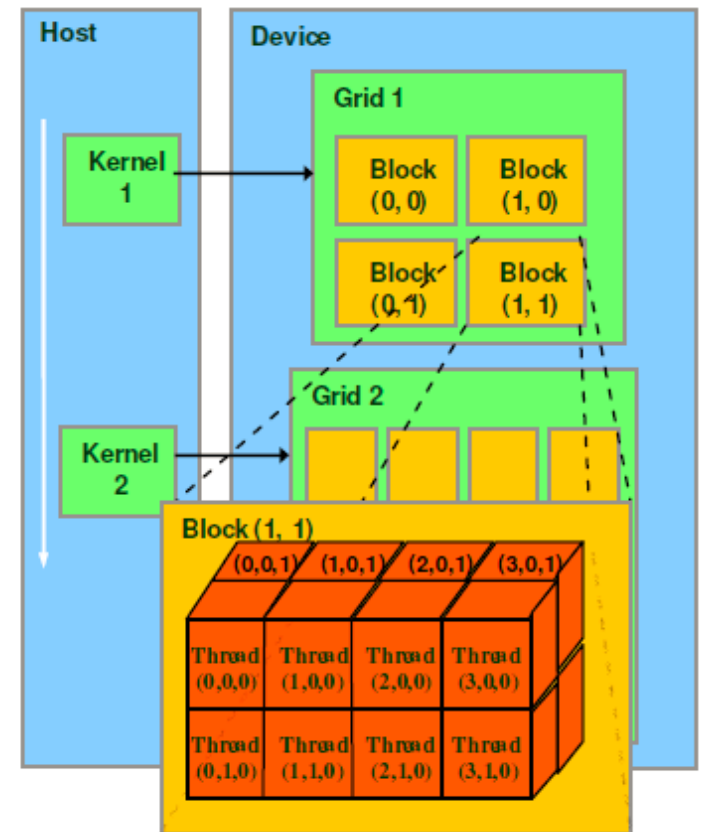
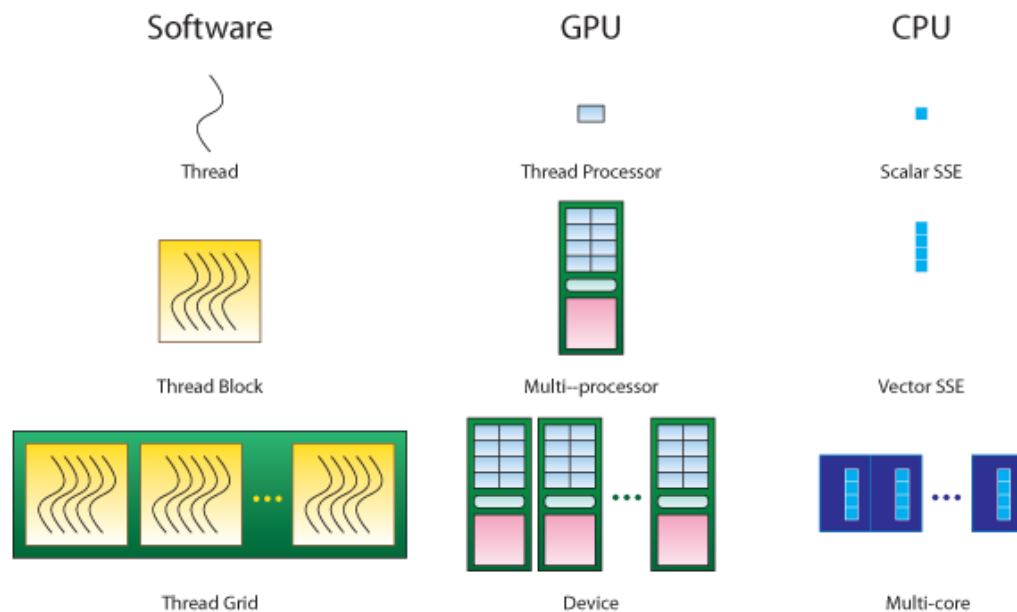
# Cuda Model and Processing Flow

- What is **CUDA**?
  - It is a set of C/C++ extensions to enable the **GPGPU** computing on **NVIDIA GPUs**
  - Dedicated APIs allow to control almost all the functions of the graphics processor
- Three steps:
  - 1) copy data from **Host** to **Device**
  - 2) copy **Kernel** and execute
  - 3) copy back results



# Software and hardware

- CUDA is a realization of the heterogeneous computing paradigm: CPU for serial part and GPU for parallel part
- CUDA maps the hardware architecture to high level software programming



*HOST: CPU*  
*DEVICE: GPU*

# Structure of a Program

```
#include <stdio.h>

#define N 1048576
#define THREADS_PER_BLOCK 512

void RandomVector(int *a, int nn){
    for (int i=0;i<nn;i++){
        a[i]=rand()%100+1;
    }
}

//kernel
__global__ void VecAddGpu(int *a, int *b, int *c){
    int index = threadIdx.x + blockIdx.x*blockDim.x;
    c[index] = a[index]+b[index];
}

int main(void) {
    int *h_a, *h_b, *h_c;
    int *d_a, *d_b, *d_c;
    int size = N*sizeof(int);

    float time;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    //Alloc in Host (and filling)
    h_a = (int *)malloc(size);
    h_b = (int *)malloc(size);
    h_c = (int *)malloc(size);
    RandomVector(h_a, N);
    RandomVector(h_b, N);

    //Alloc in Device
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    //Copy input vectors form host to device
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    //start time
    cudaEventRecord(start);

    //Launch Kernel on GPU
    VecAddGpu<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(&d_a, &d_b, &d_c);
    cudaDeviceSynchronize();

    //stop time
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);

    //Copy back the results
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    //Print Result
    // for(int i=0;i<N;i++){
    //     printf ("%d) h_a:%d h_b:%d h_c:%d\n", i, h_a[i], h_b[i], h_c[i]);
    // }

    //print time
    printf("Time: %3.5f ms\n", time);

    //Cleanup
    free(h_a);
    free(h_b);
    free(h_c);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}
```

**KERNEL - DEVICE**

**MAIN - HOST**

**COPY DATA H→D**

**LAUNCH KERNEL**

**COPY DATA D→H**

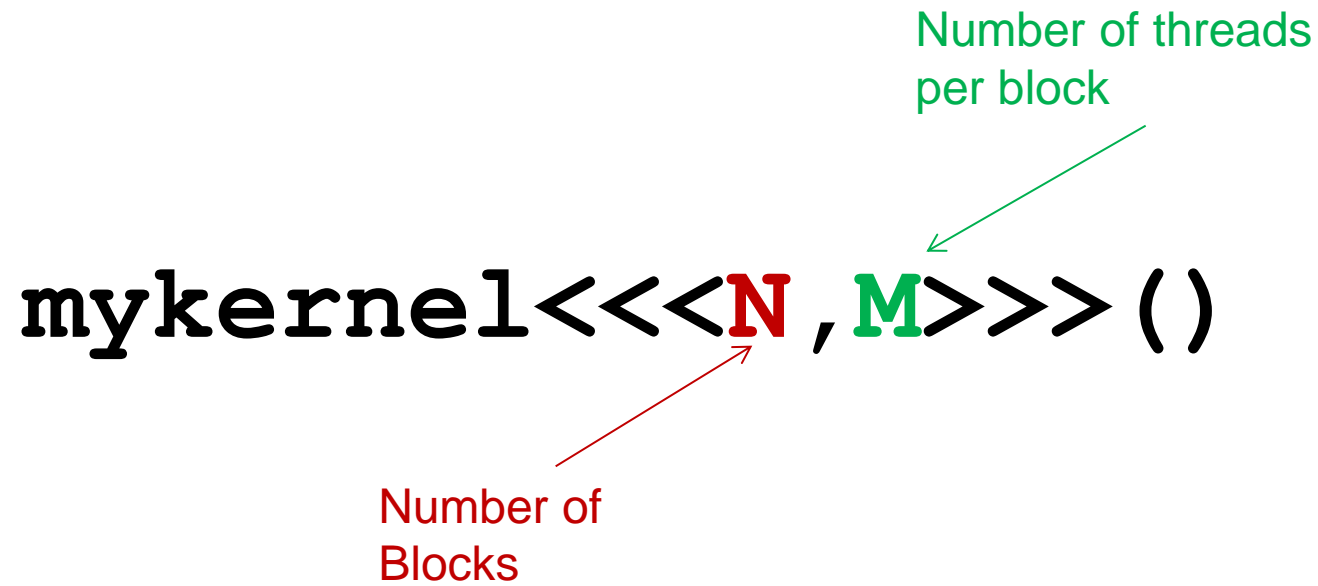


# Kernel call

mykernel<<<**N**, **M**>>> ()

Number of threads  
per block

Number of  
Blocks

The diagram shows a kernel call 'mykernel<<<N, M>>> ()'. The 'N' is red and has a red arrow pointing to it from the text 'Number of Blocks' below. The 'M' is green and has a green arrow pointing to it from the text 'Number of threads per block' above.

- Keep in mind: the number of threads per block is limited



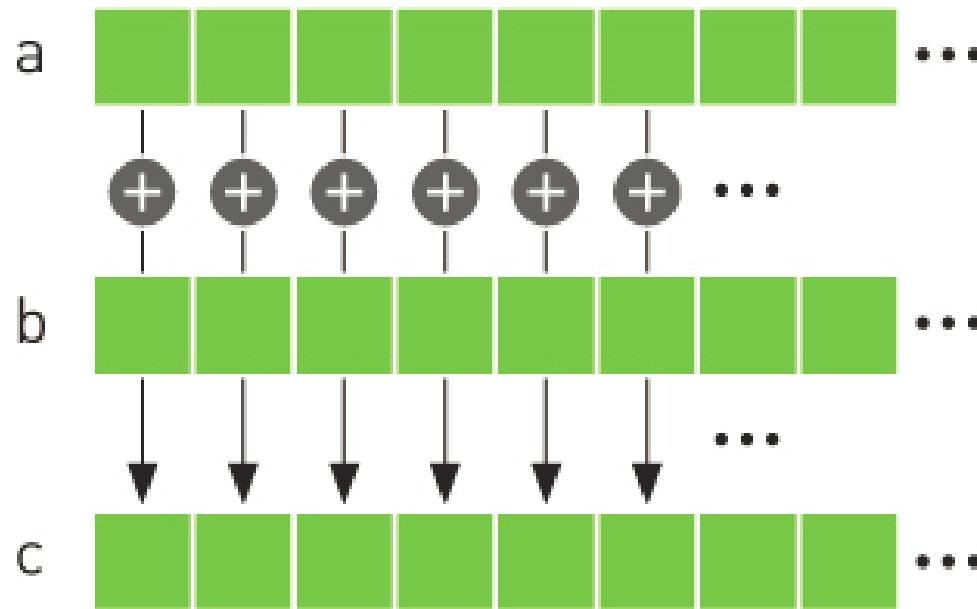
- Host “Hello World!”



- Parallel “Hello World!”



# Vector Sum



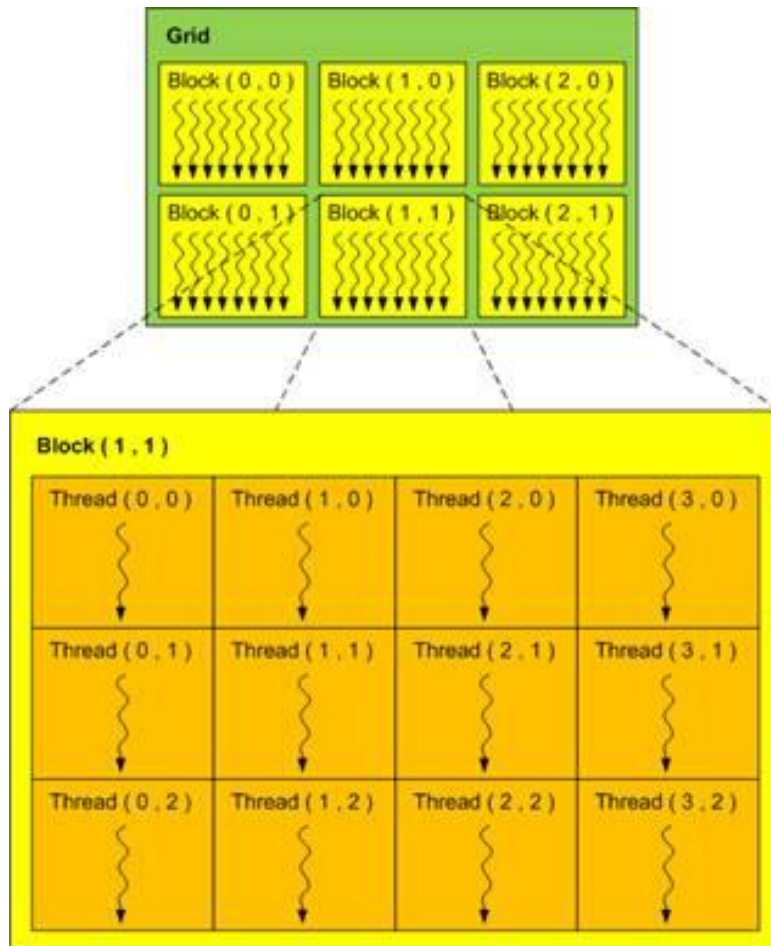
- Serial code



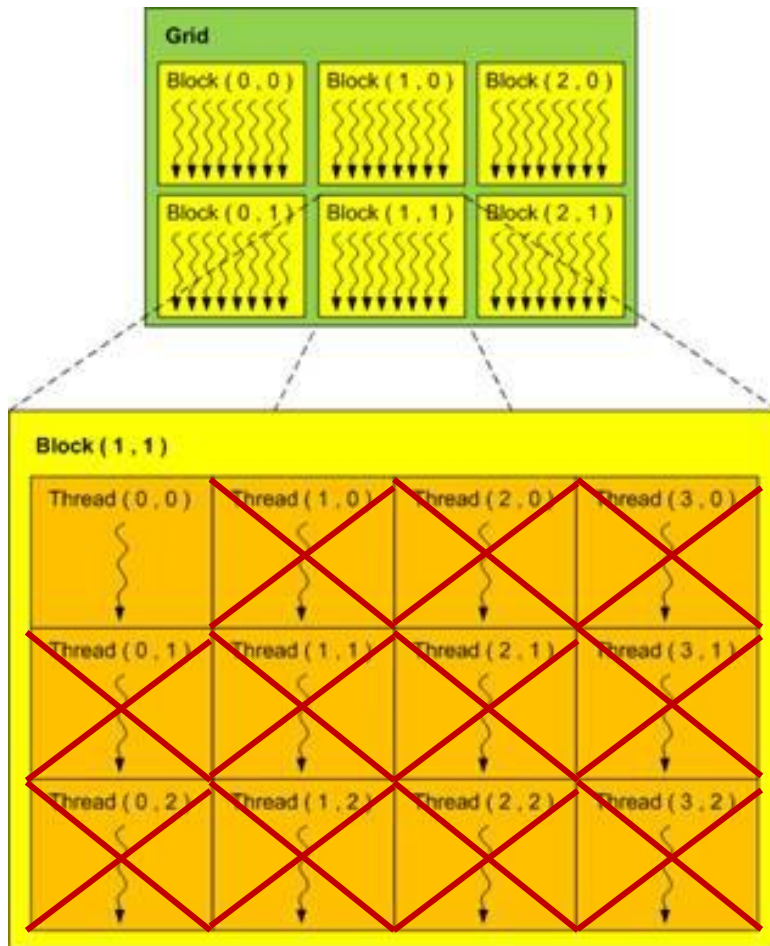
- VecAdd with Blocks



- threadIdx.x, blockIdx.x
- cudaMalloc(), cudaMemcpy(),  
cudaFree()



- We are using a big grid (1048576), but each block uses only one thread
  - A block doesn't correspond exactly to on SM



- Very inefficient
- The GT750 has a limited number of Multiprocessors (4)
- Roughly speaking: only 4 threads are concurrent

- 
- With threads

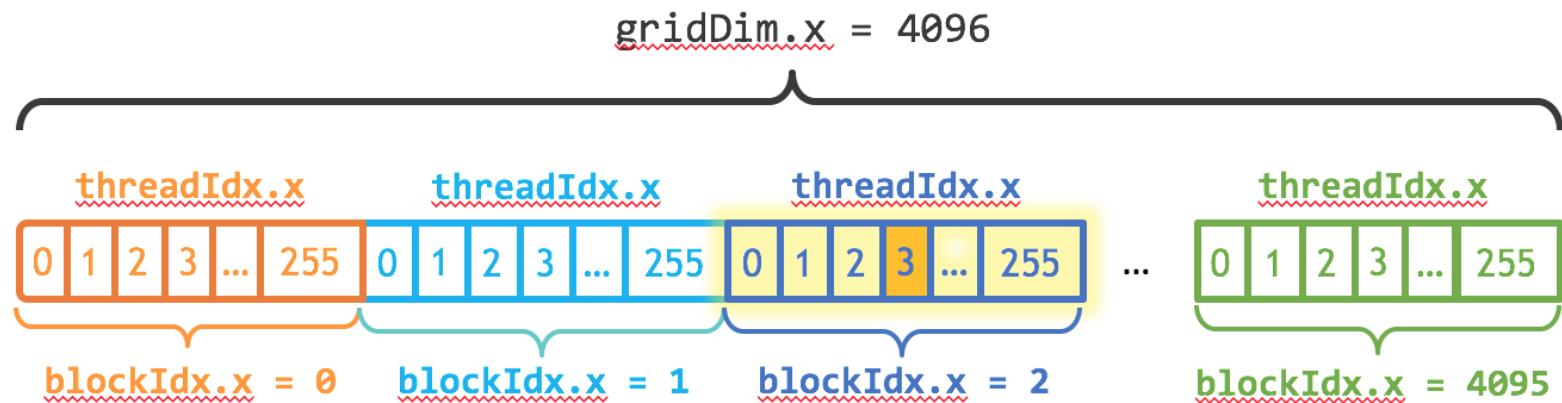


- Factor 500! Suspect... → printout, cuda-gdb, error checking



# Thread index

- The right way is to use blocks & threads
- ... with the correct indexing



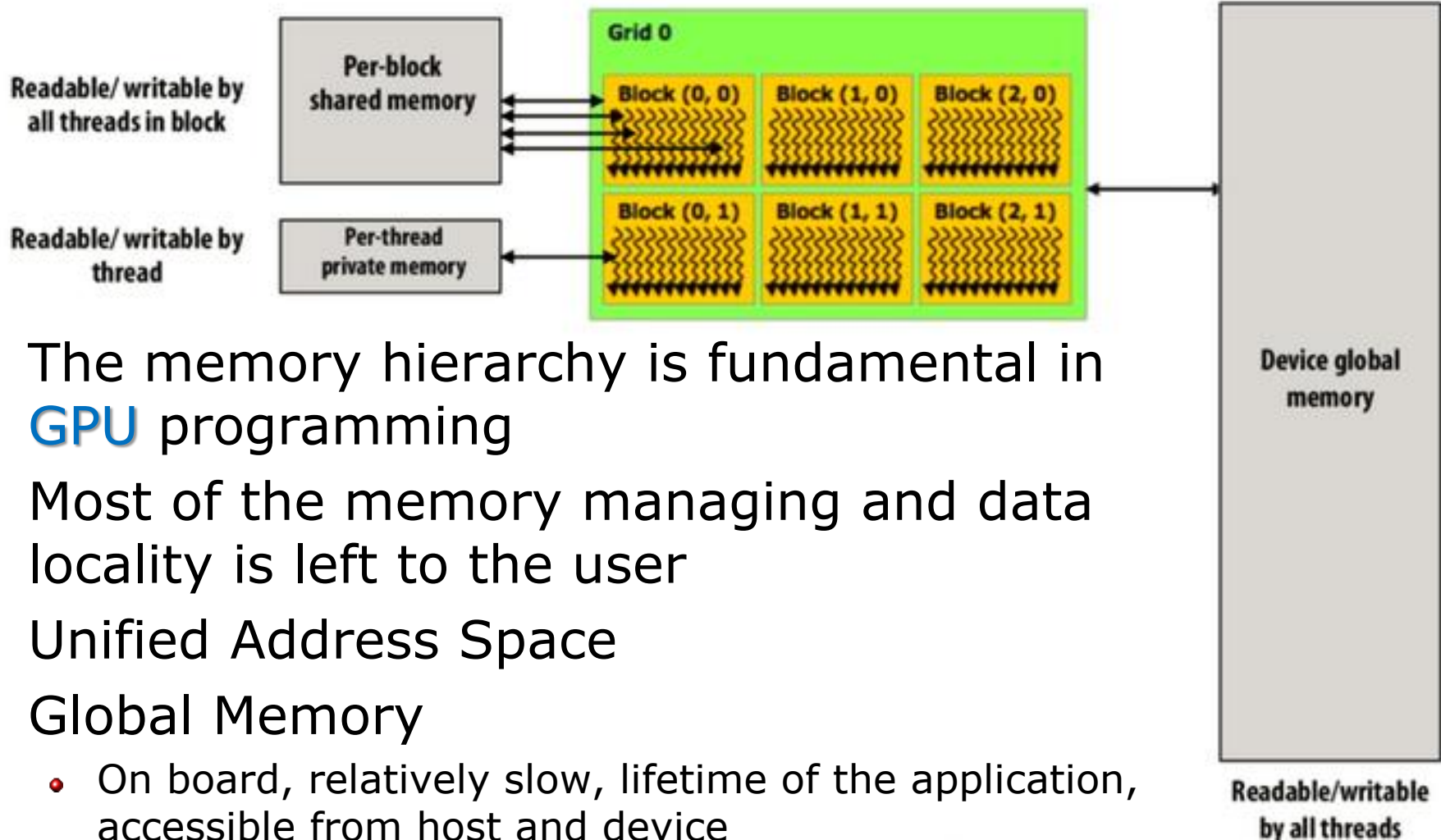
$$\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

$$\text{index} = (2) * (256) + (3) = 515$$

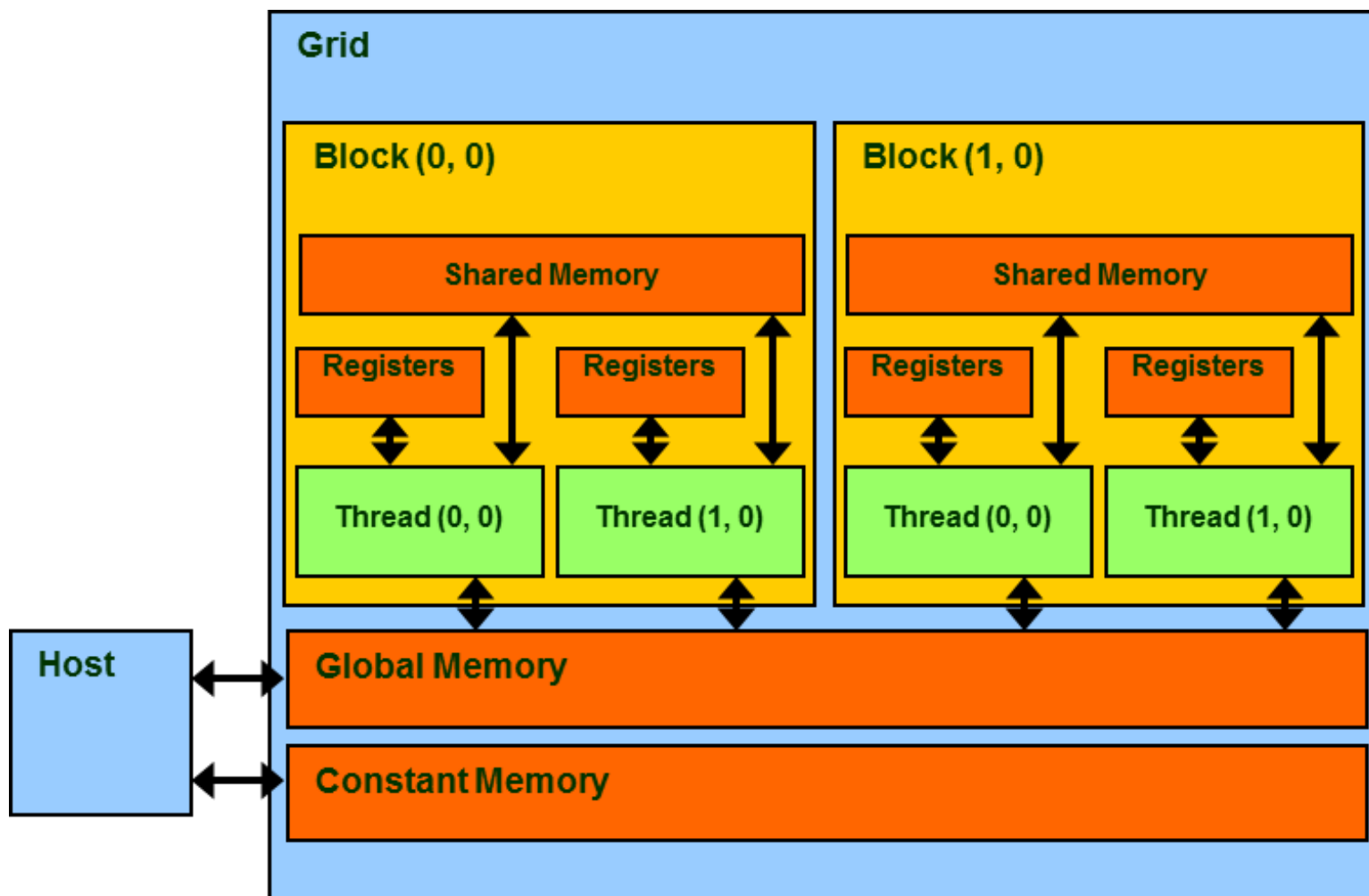
# Threads , Blocks & Warps

---

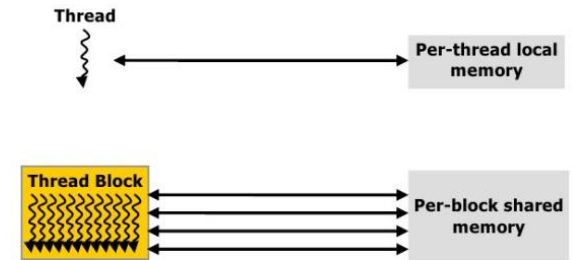
- Threads and Blocks are not equivalent
    - The main difference is that the threads can “communicate”
  - The instruction set is pipelined in the hardware → the most basic unit of SIMD scheduling is a “Warp”
  - A warp consists of 32 threads
  - If the number of threads is multiple of 32 then no **divergence**
  - The warp is also the natural unit to access memory
-



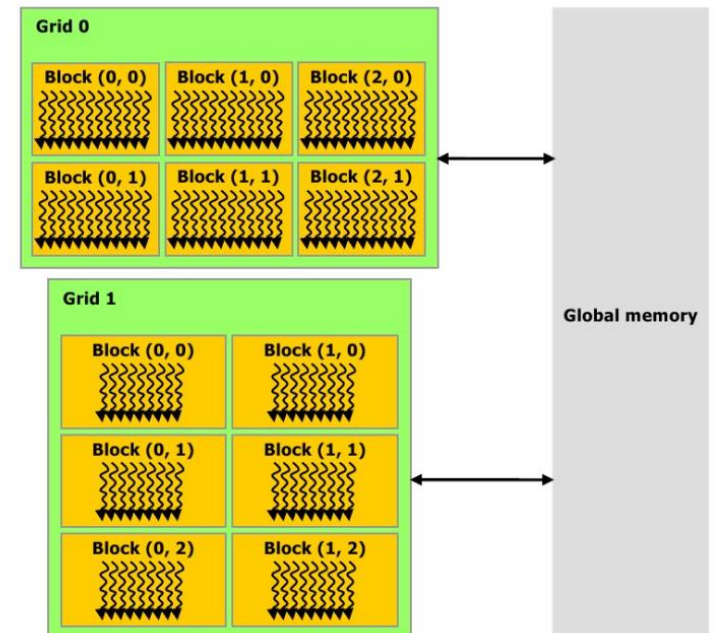
- The memory hierarchy is fundamental in **GPU** programming
- Most of the memory managing and data locality is left to the user
- Unified Address Space
- Global Memory
  - On board, relatively slow, lifetime of the application, accessible from host and device
- Shared memory/registers
  - On Chip, very fast, lifetime of blocks/threads, accessible from kernel only



Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

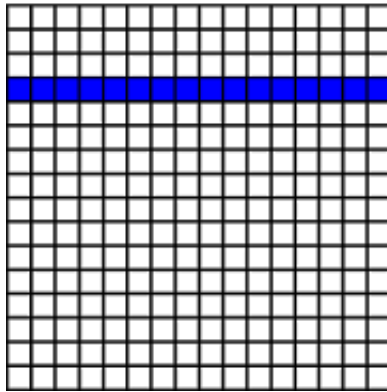


- Automatic variables reside in a register
  - Except per-thread arrays that reside in global memory

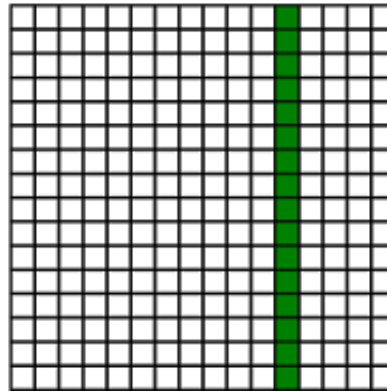


# Matrix Multiplication: naive implementation

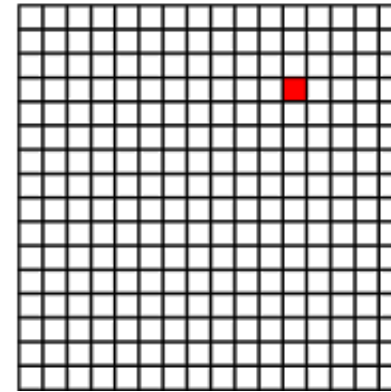
A



B



C



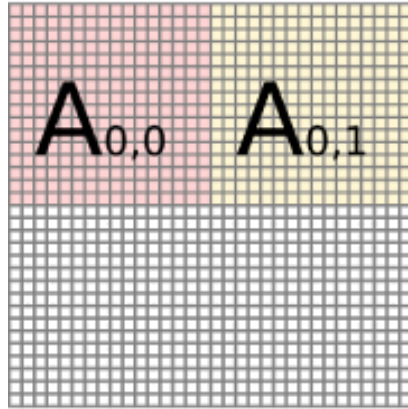
- One thread to compute one element of matrix C. Each thread loads one row of matrix A and one column of matrix B from global memory, and store the result back to matrix C in the global memory.
- Number of operations:  $M \times N \times K \times 2$
- Number of memory access:  $M \times N \times K \times 2$  words  $\rightarrow$   $4 \times M \times N \times K \times 2$  bytes
- Computation to memory ratio =  $\frac{1}{4}$   $\leftarrow$  memory bounded

HANDS ON

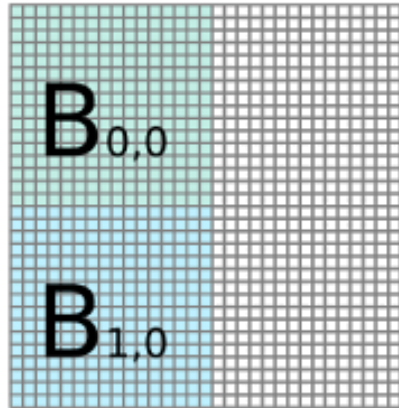


# Matrix multiplication: tiled implementation

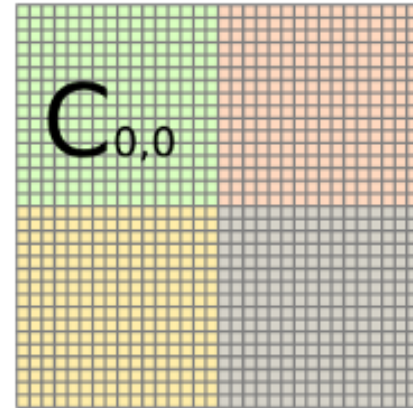
A



B



C



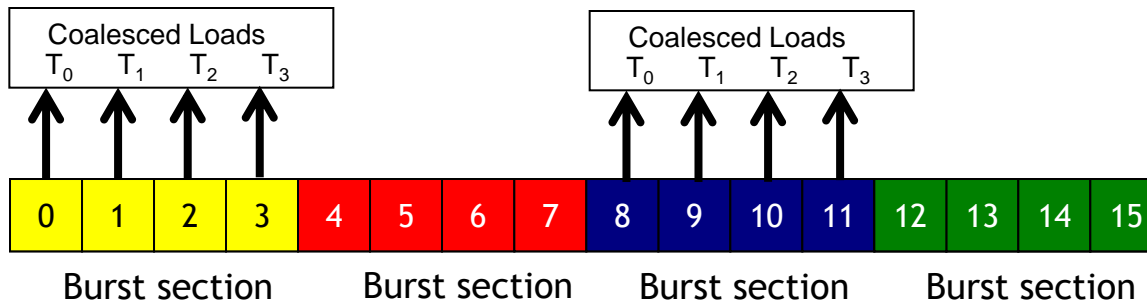
- One thread block computes one tile of matrix C. One thread in the thread block computes one element of the tile.
- In each iteration, one thread block loads one tile of A and one tile of B from global memory to shared memory, performs computation, and stores temporal result of C in register. After all the iteration is done, the thread block stores one tile of C into global memory.
- If the tile size is B, the amount of global memory access is  $2 * M * N * K / B$ , The “computation-to-memory” ratio is  $B/4$  (flop/byte)

HANDS ON



## Further optimization: Memory access

- Global memory coalescing

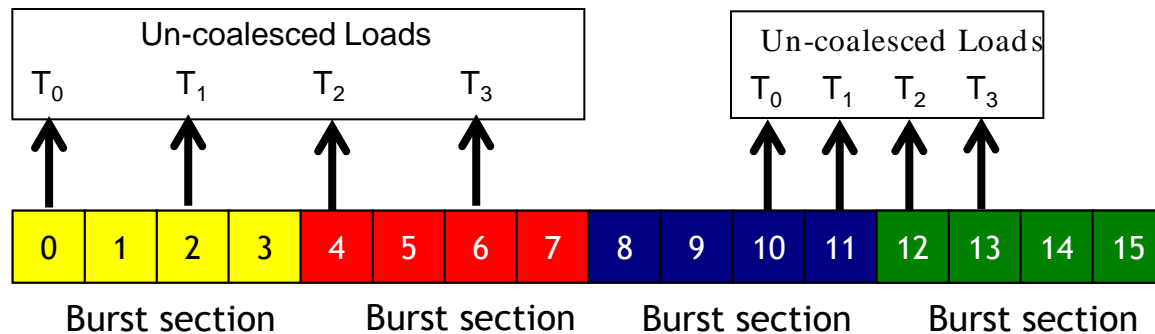


- When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.



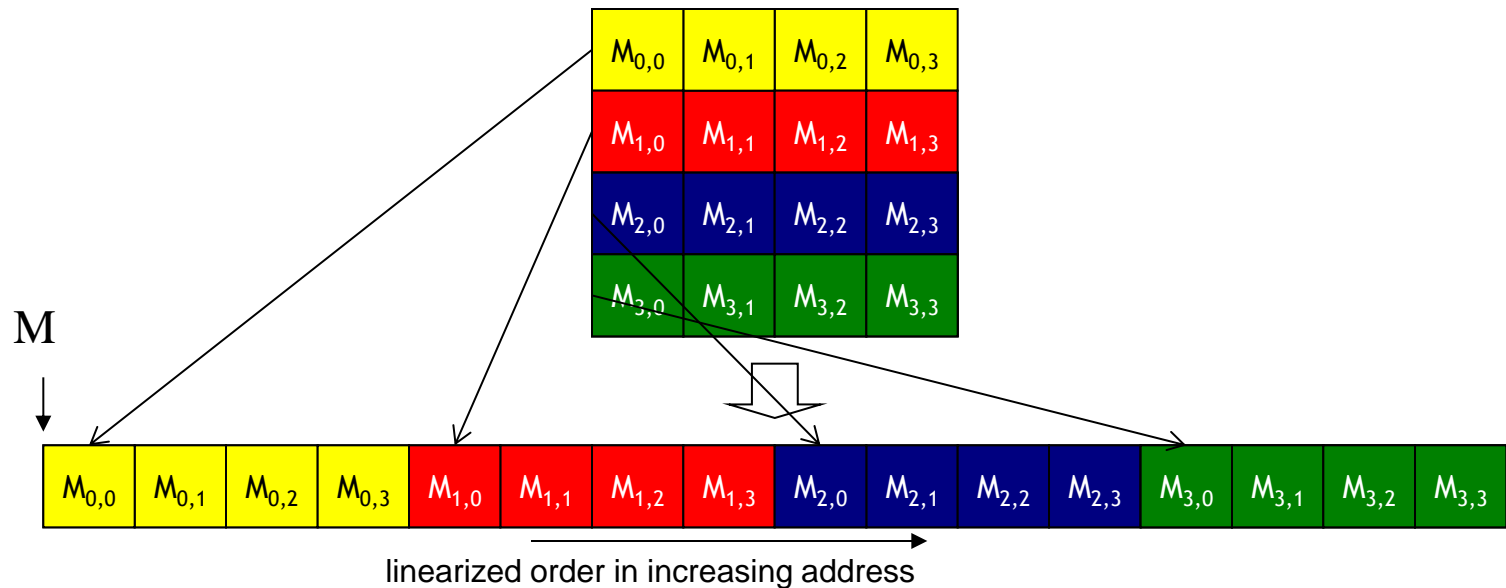
## Further optimization: Memory access

- Global memory coalescing



- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

# Further optimization: memory access



- In C the 2D arrays are linearized in “row-major” (opposite in Fortran)
- In Matrix multiplication transpose one of the two matrixes

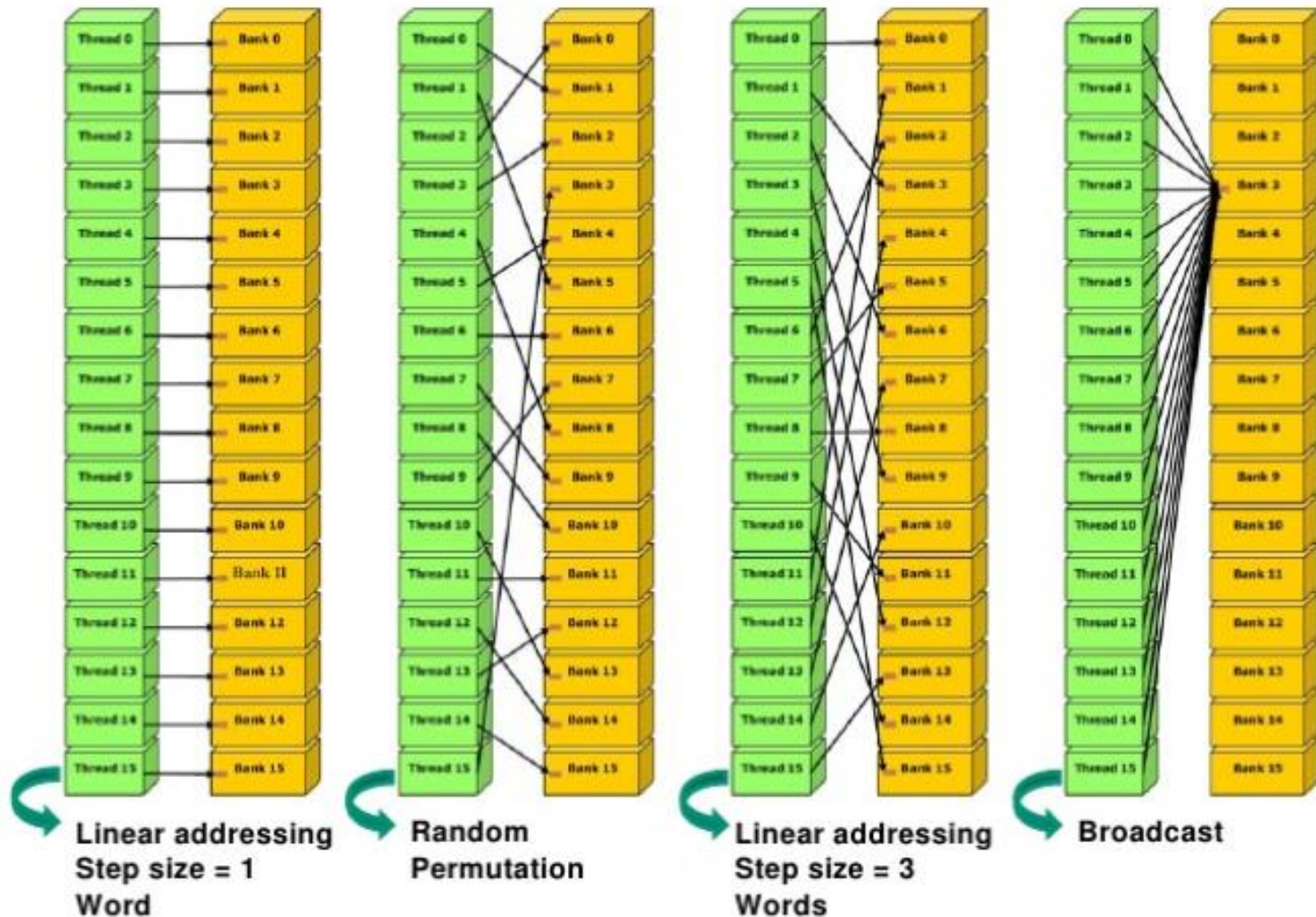
## Further optimization: shared memory

---

- To achieve high memory bandwidth for concurrent accesses the Shared Memory is organized in banks that can be accessed simultaneously :
  - 16 banks in older GPU, 32 banks in modern GPU
- If multiple addresses of a memory request map to the same memory bank, the accesses are serialized

# Further optimization: shared memory

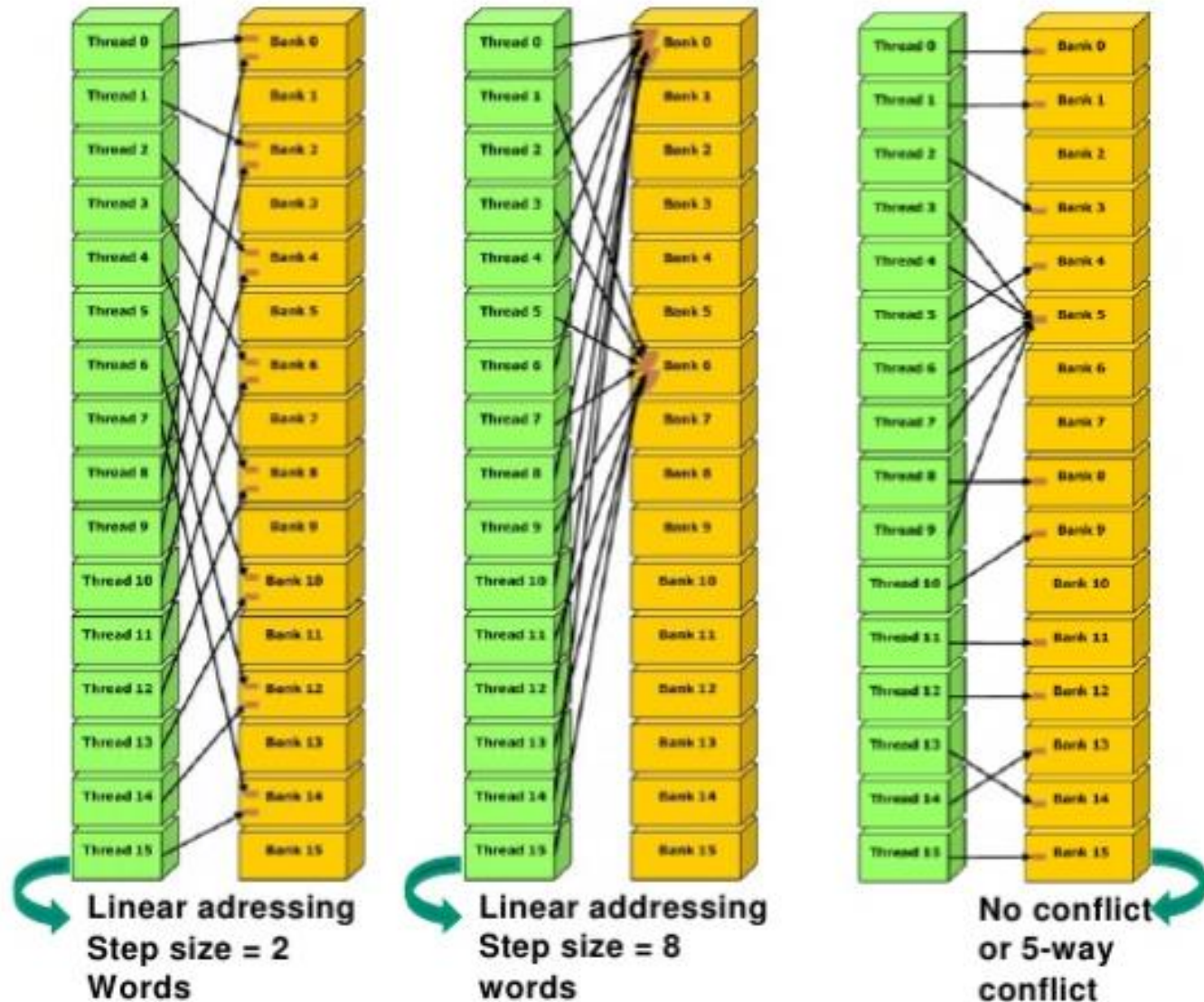
- To combine
- If re



or  
n  
J

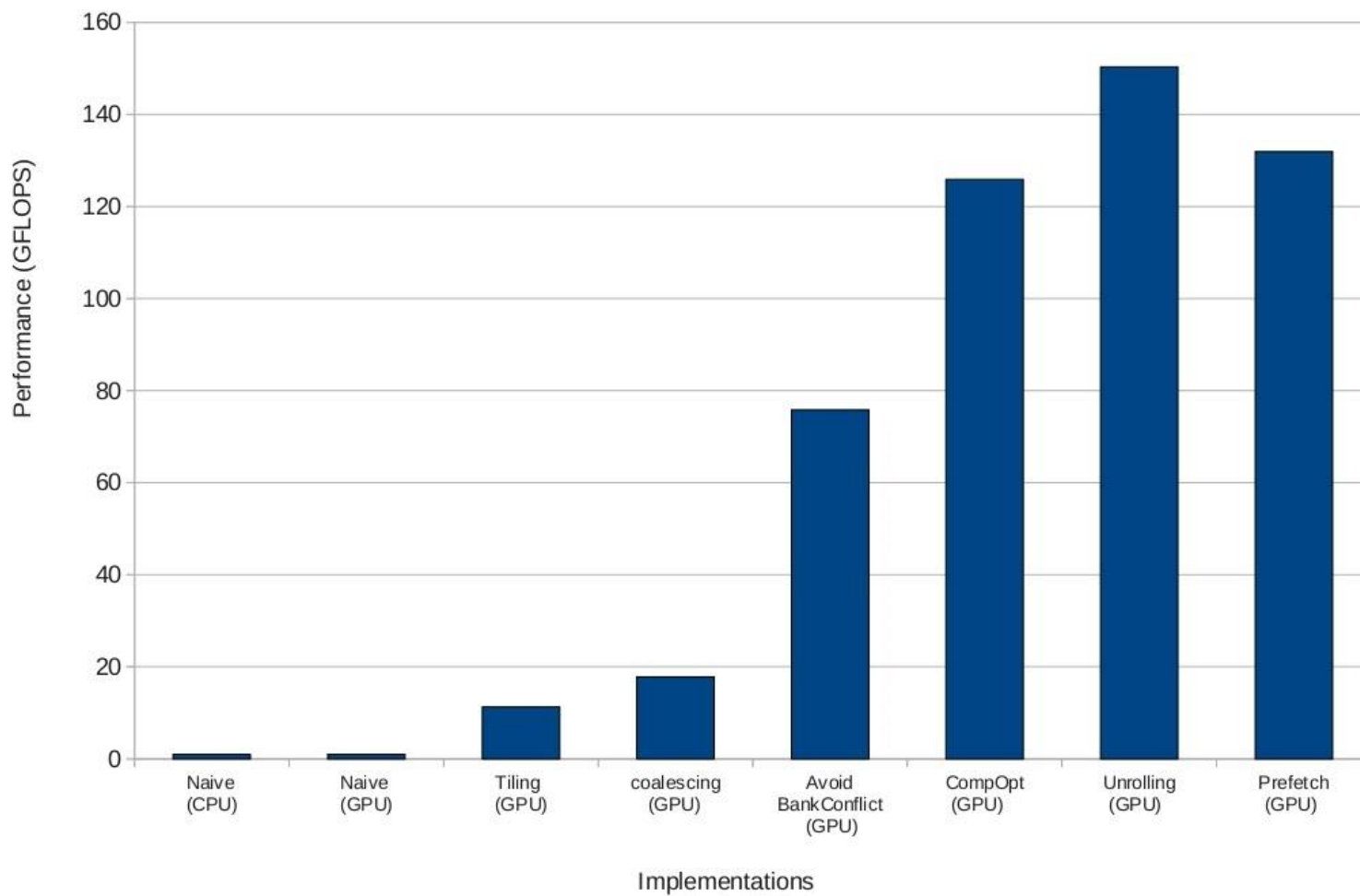
# Further optimization: shared memory

- To correct memory bank conflicts
- If the number of threads is a multiple of the number of banks



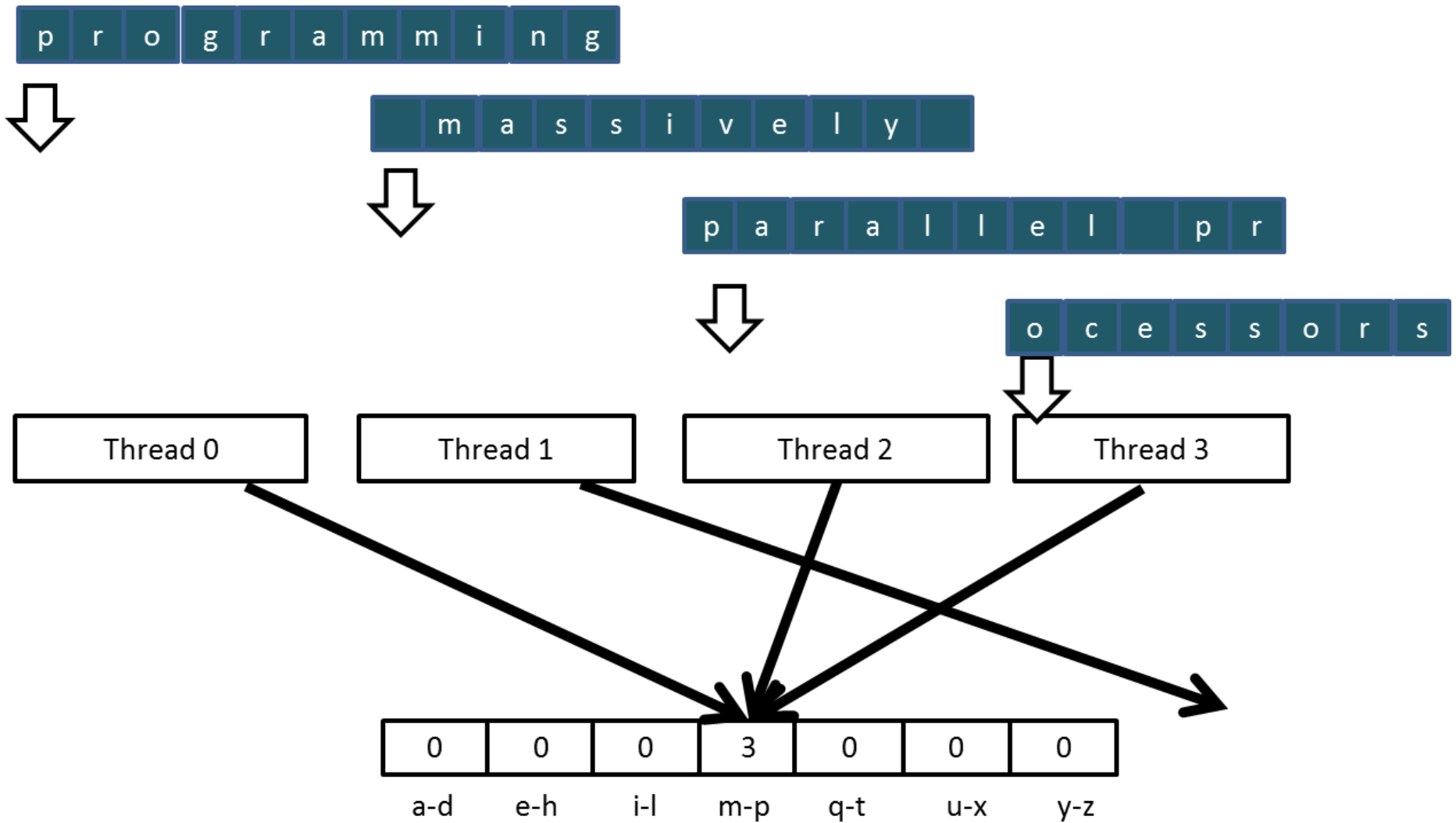
for  
an  
PU

Performance of different implementation methods  
measured on 8800 GT with matrix size of 4096 x 4096





# Histograms



p r o g r a m m i n g



m a s s i v e l y



p a r a l l e l p r



o c e s s o r s

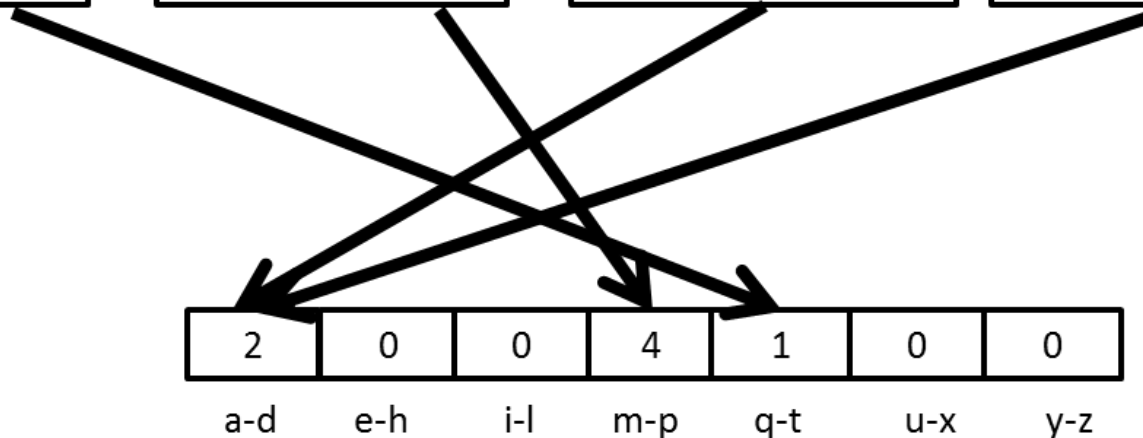


Thread 0

Thread 1

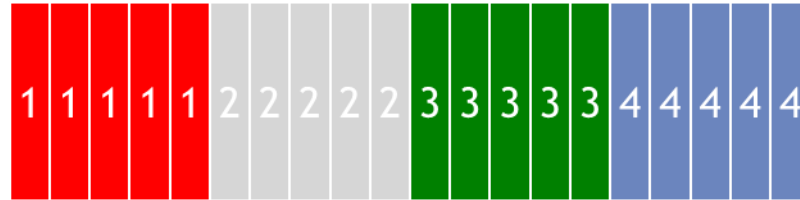
Thread 2

Thread 3

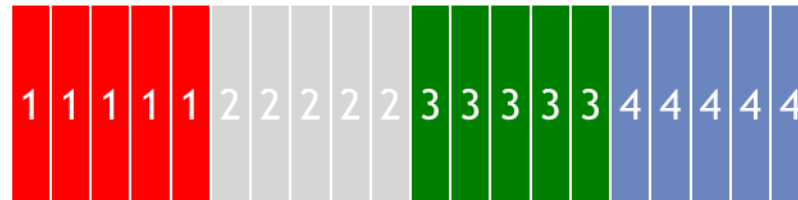




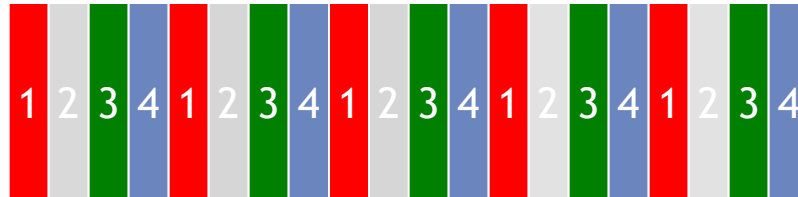
- Sectioned partitioning results in poor memory access efficiency
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized

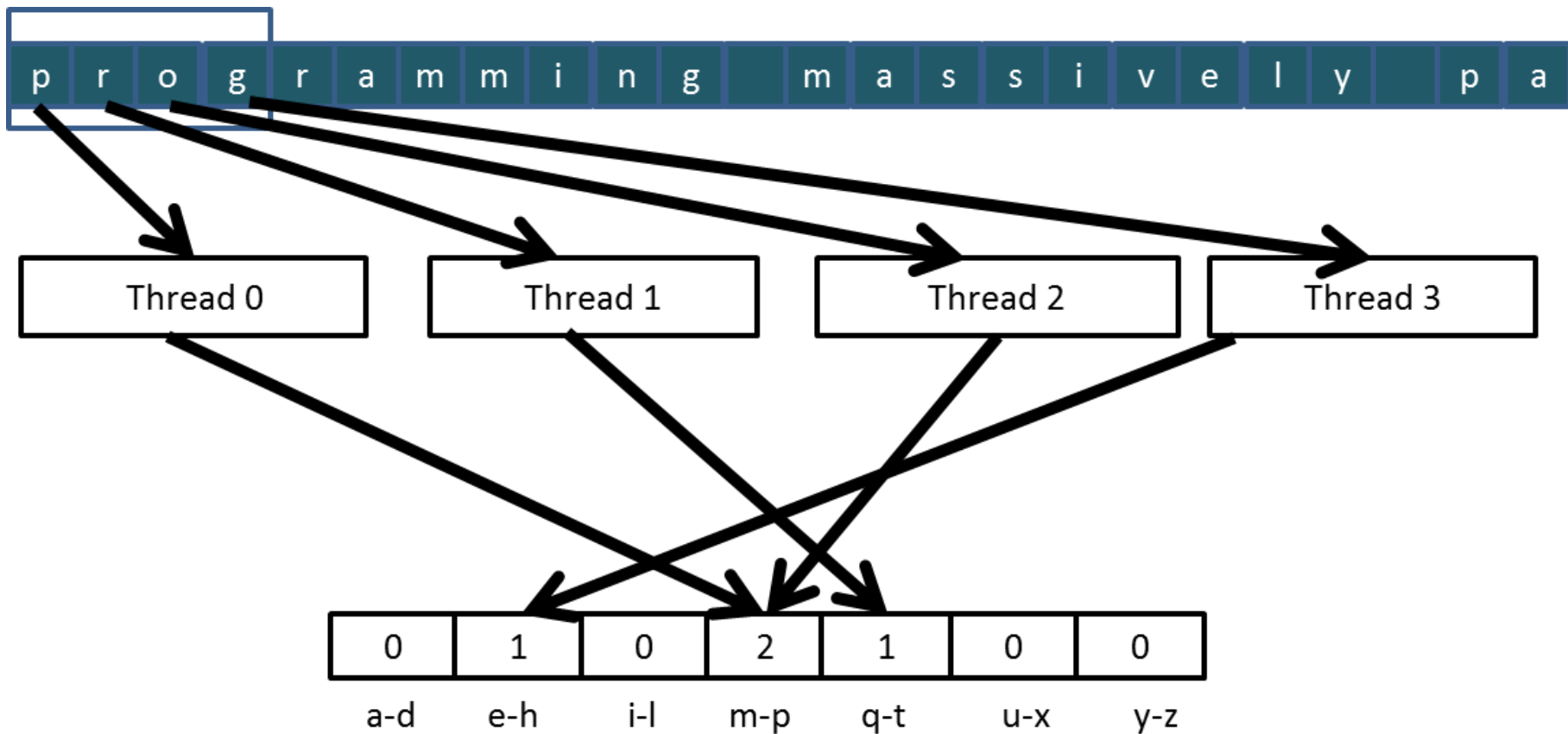


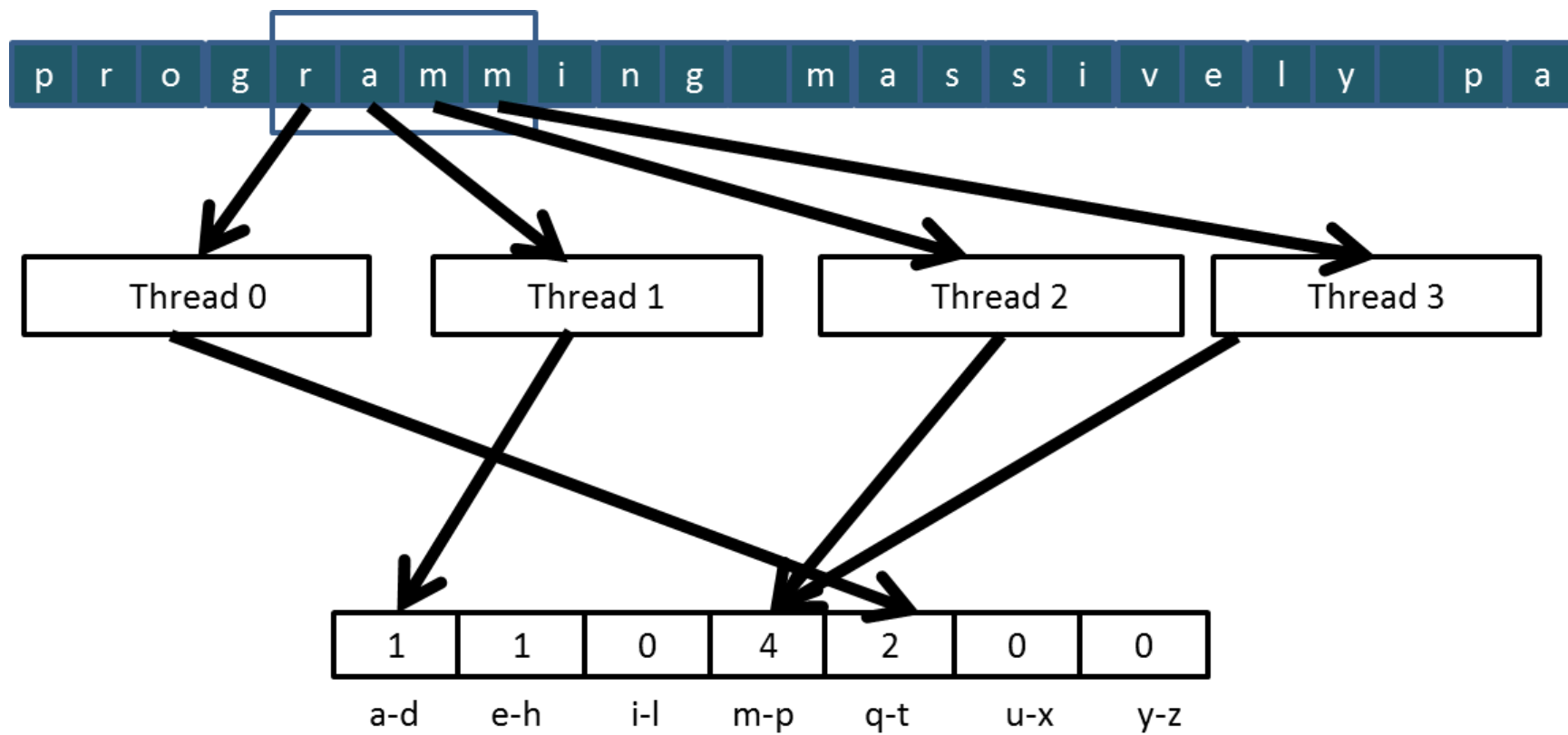
- Sectioned partitioning results in poor memory access efficiency
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized



- Change to interleaved partitioning
  - All threads process a contiguous section of elements
  - They all move to the next section and repeat
  - The memory accesses are coalesced







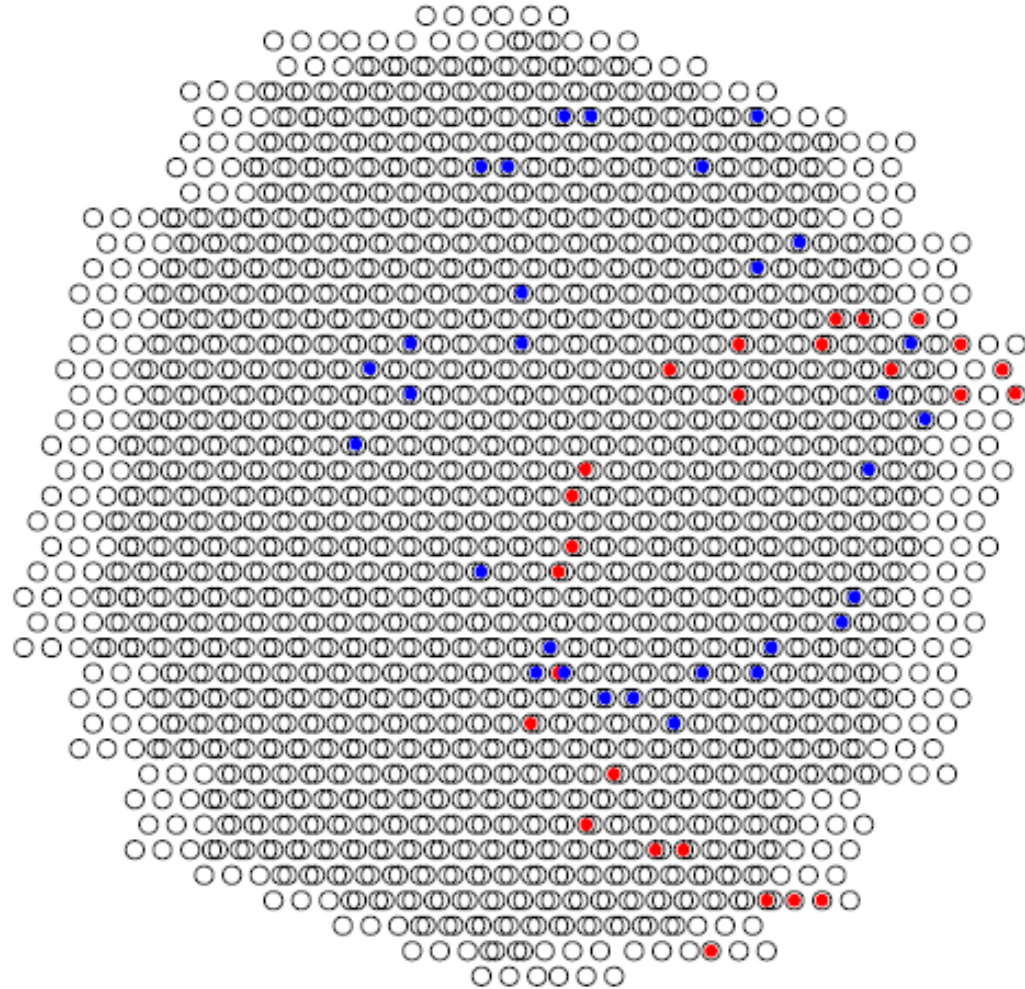
# Histograms: data race

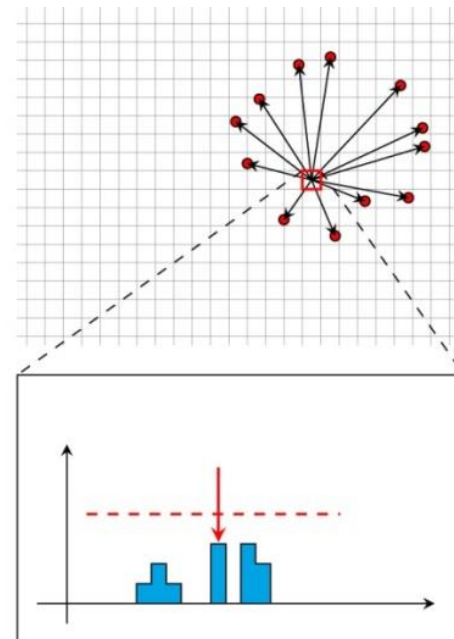
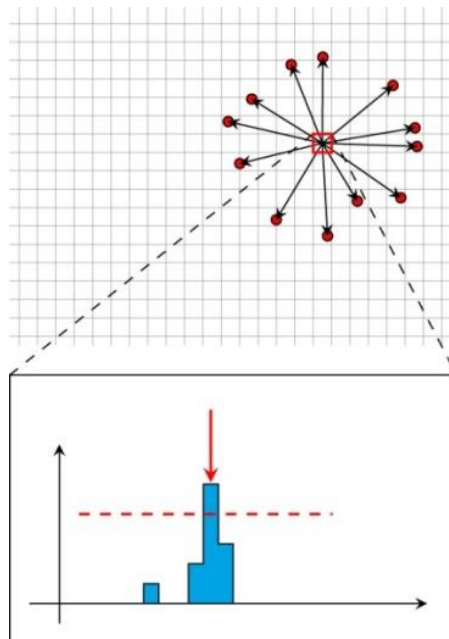
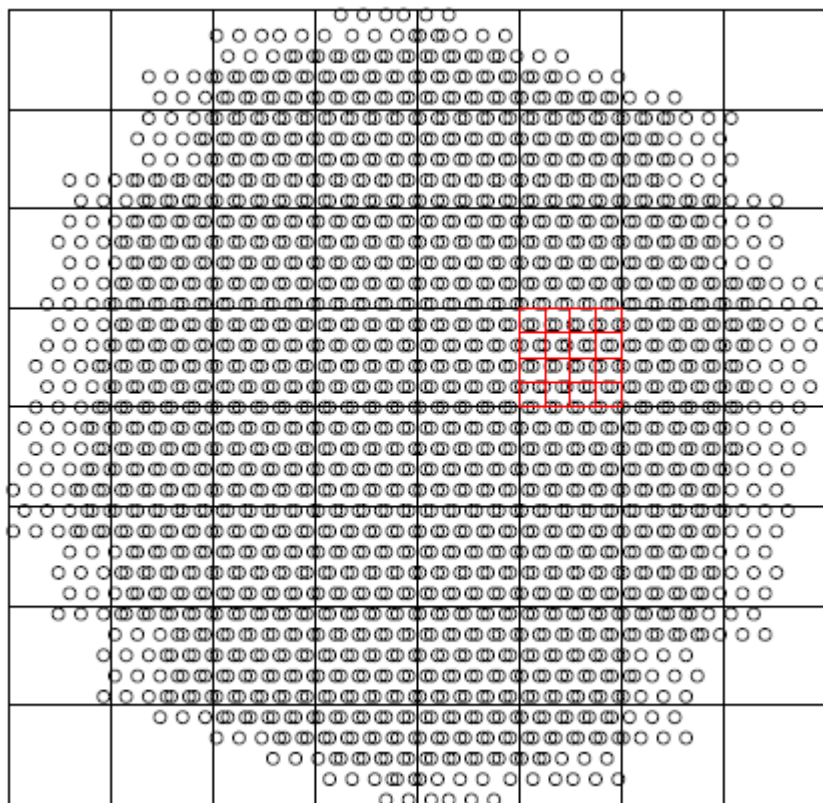
---

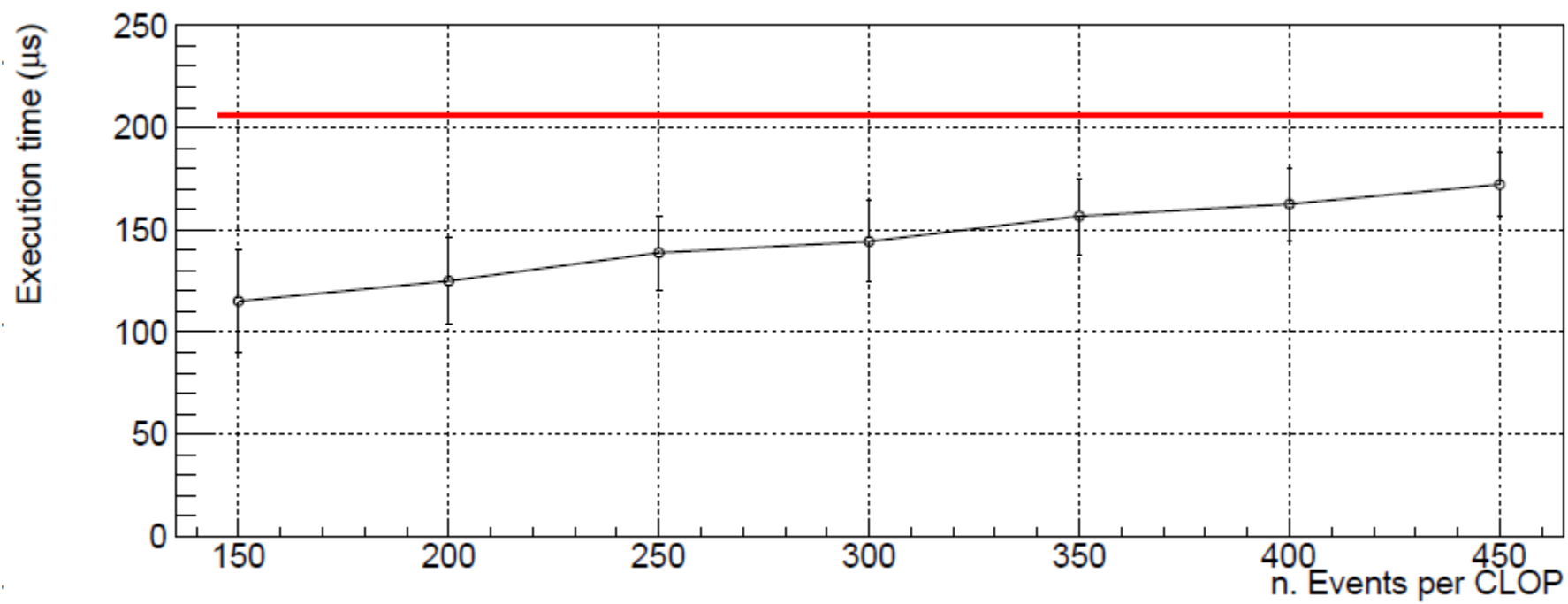
- Each thread must update the previous value of the bin
- This is not compatible with parallel threads  
→ data race
- The hardware ensures that no other threads can perform another read-modify-write operation on the same location until the current **atomic operation** is complete
  - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)



# Histograms in HEP

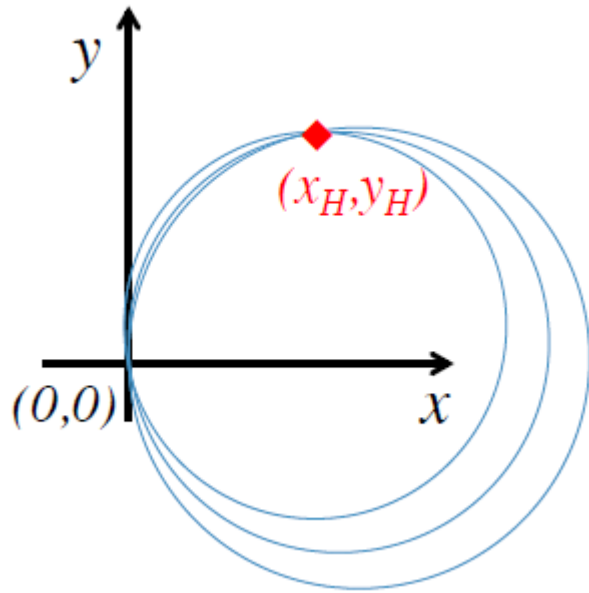








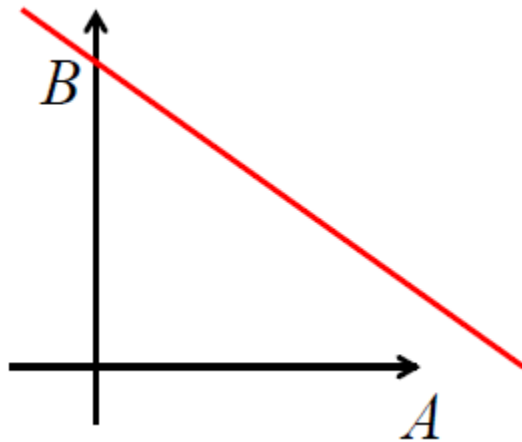
# Hough Transform in GPU



- There are infinite rings passing through (x<sub>H</sub>, y<sub>H</sub>) and (0,0)

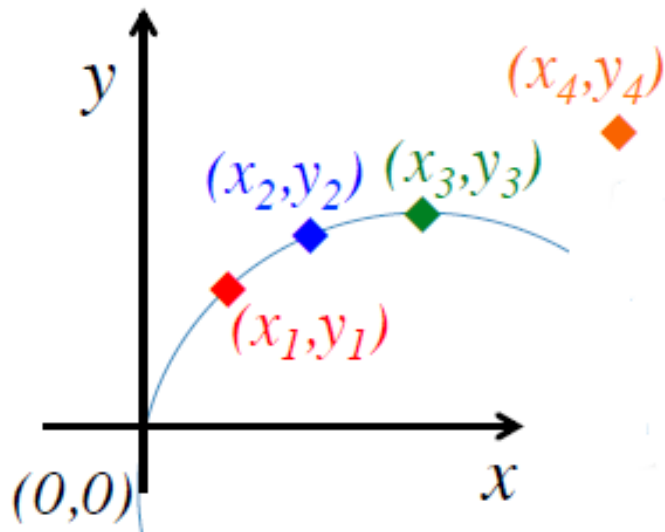
$$x_H^2 + y_H^2 - 2Ax_H - 2By_H = 0$$

- All the infinite rings are represented by a straight line in the parameters space

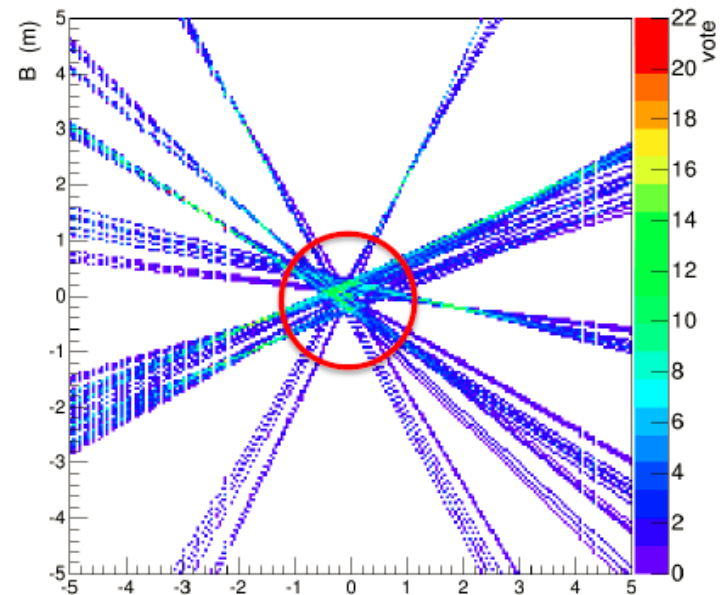
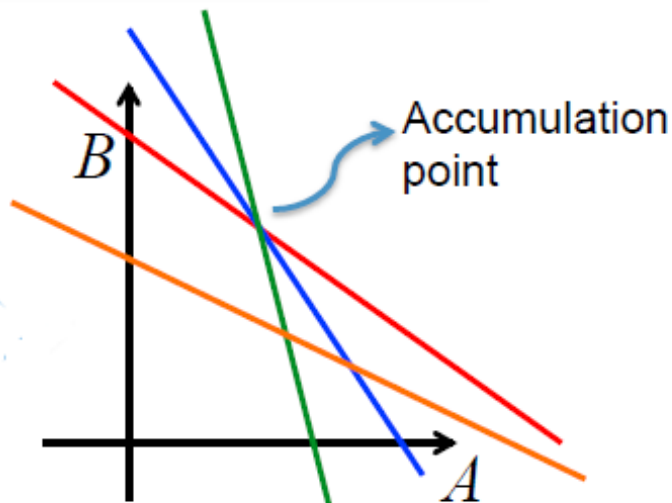


$$B = \frac{x_H^2 + y_H^2 - 2Ax_H}{2By_H}$$

# Hough Transform in GPU



- Discretize the Hough space
- Look for accumulation point
- ... the life is not so easy

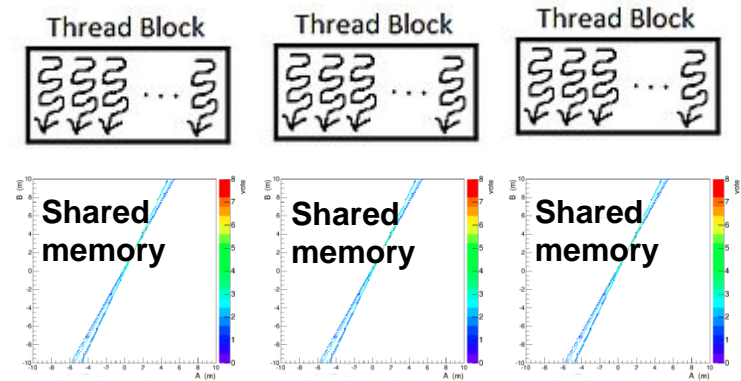


[Rinaldi GPUinHEP 2014]

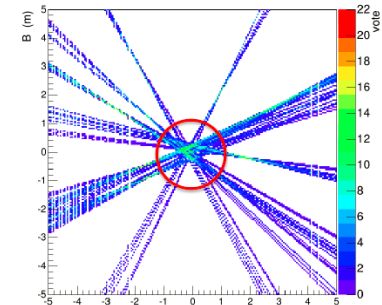
# Hough Transform in GPU

- Intrinsically parallel algorithm
  - Each hit generates an independent line in the Hough space
- Two kernels:
  - Voting procedure
  - Extract result
- Voting procedure
  - 1 warp per SM
  - Hough Space in Shared Memory
  - Atomic Add
- Extract result
  - Copy all the histograms in shared memory to global
  - Apply thresholds (or more complicated decisions)

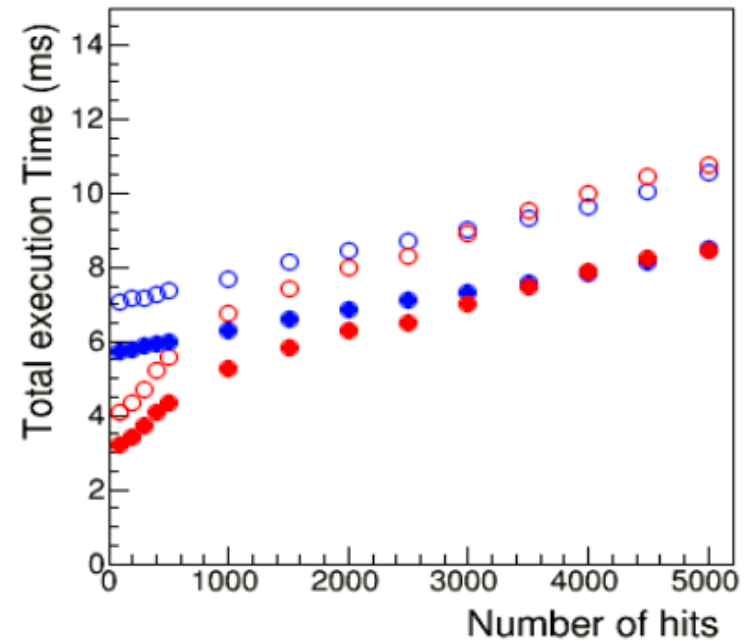
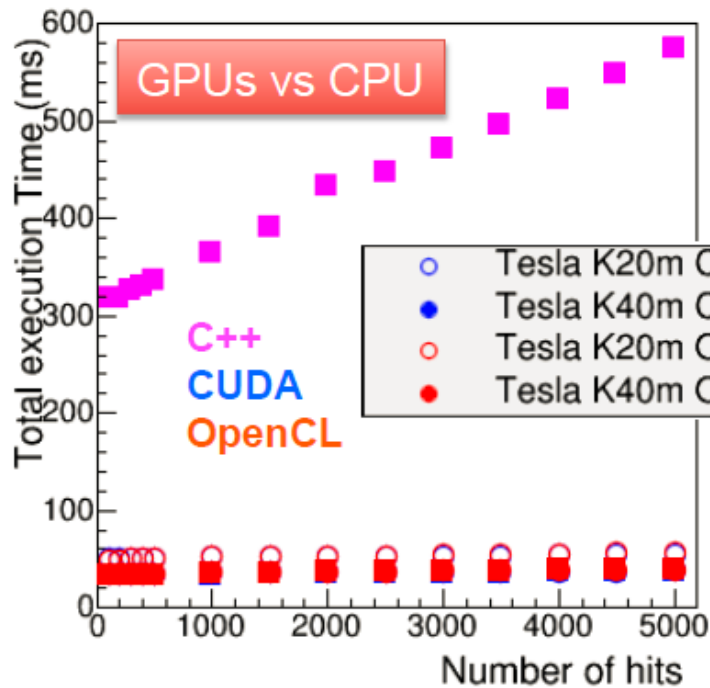
Hits Array



Merge in Global



# Hough transform results



## Tips & Tricks

---

- Avoid branches as much as possible
  - Organize data in memory in order to use coalescence
  - Avoid frequent access to global memory
  - Use atomics
  - Fill GPU with a lot of work, to hide latency
  - Use profiling tools to look for bottlenecks (Visual Profiler, Nsight, etc.)
-

# Conclusions

---

- GPU are massively parallel processors
- Often the implementation is “incremental”:
  - Load balancing (threads & blocks)
  - Memory (type & access)
  - Atomic operations
- Coding is relatively easy, optimization is relatively complicated
- A lot of documentation available on network and several good books to learn
- A lot of examples for free in cuda SDK