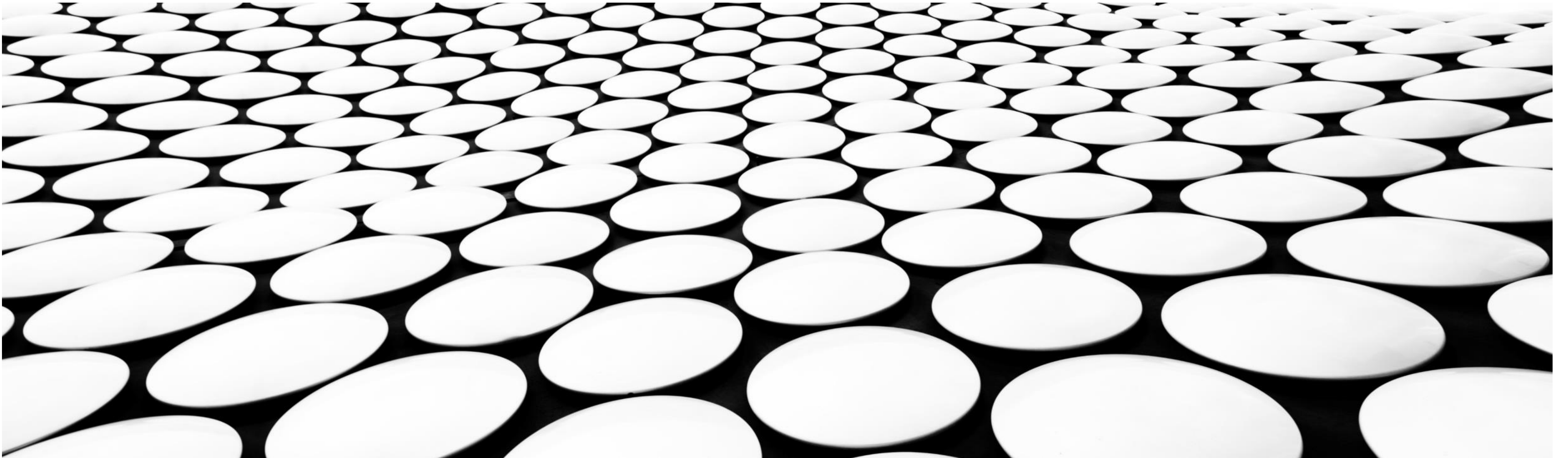
INFO-H-503 – GPGPU PROGRAMMING – 03

GAUTHIER LAFRUIT – JAN LEMEIRE

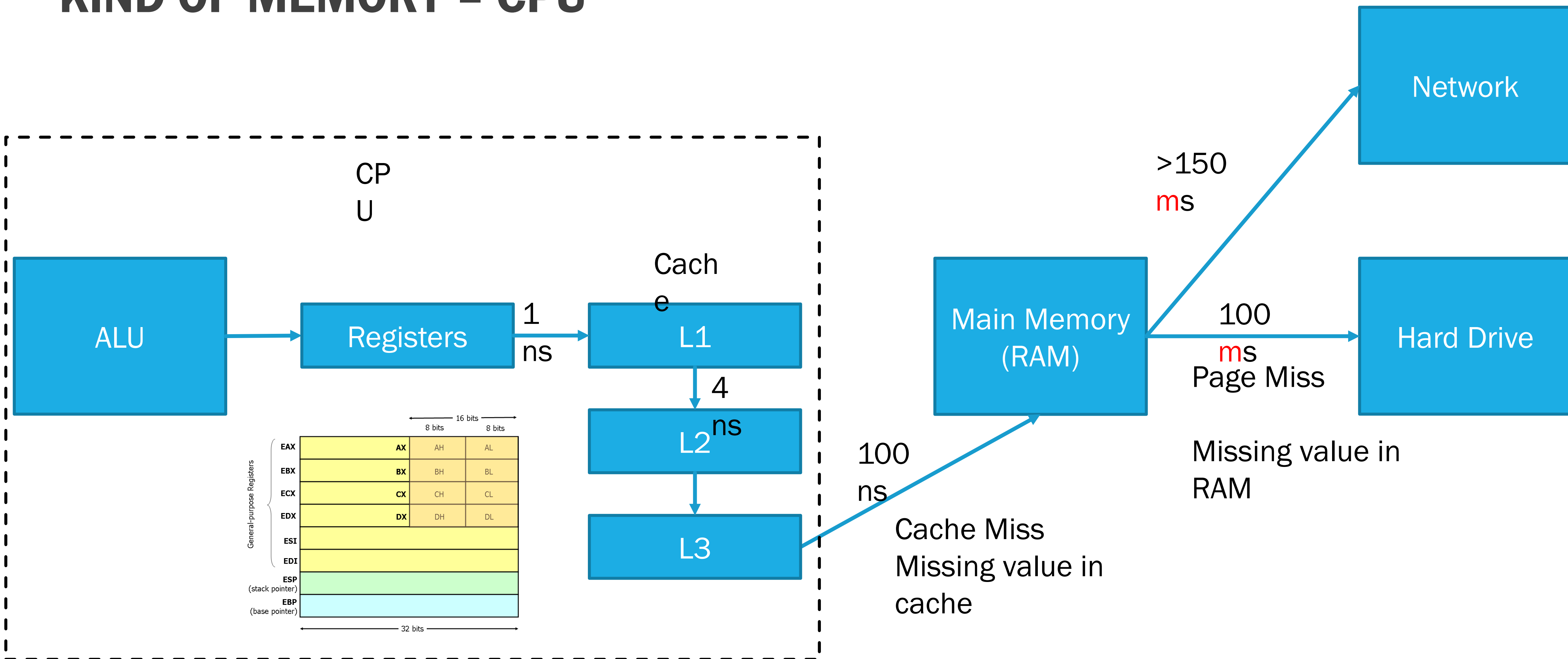
ELINE SOETENS - DANIELE BONATTO



- Last Time:
  - Vector Add
    - Occupancy
    - Profiling
    - Benchmark
- This Time:
  - Shared Memory
  - Dot product
  - Histogram

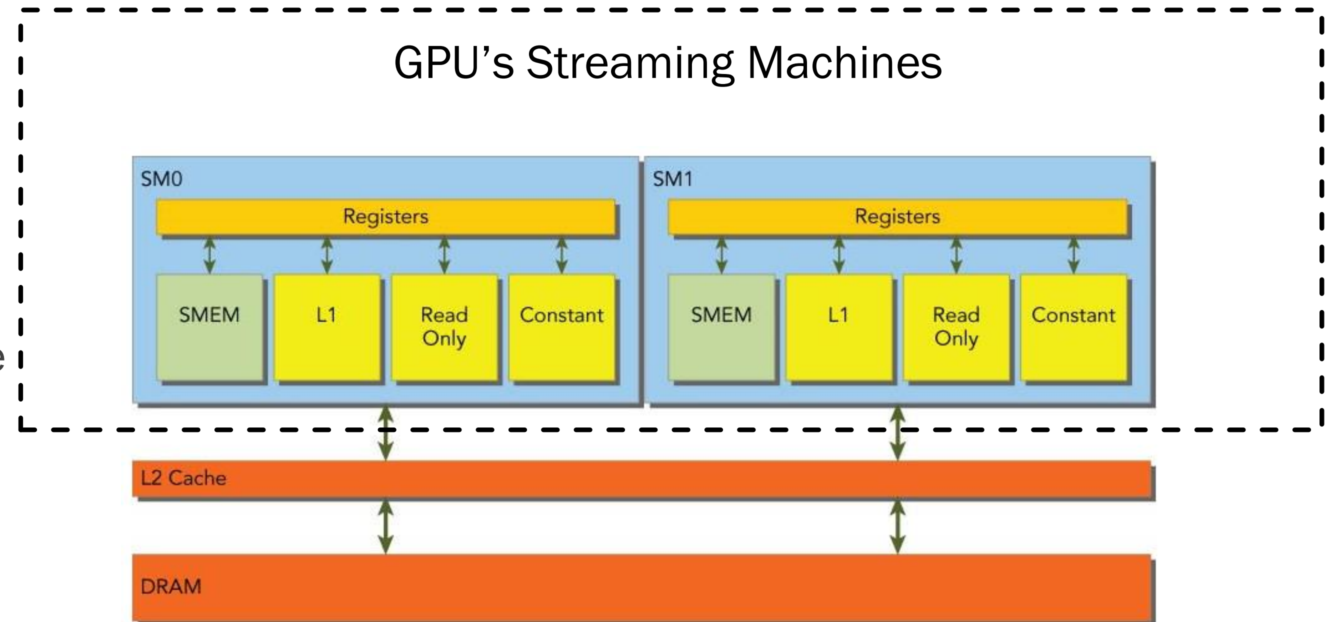
https://colin-scott.github.io/personal_website/research/interactive_latency.html

KIND OF MEMORY – CPU

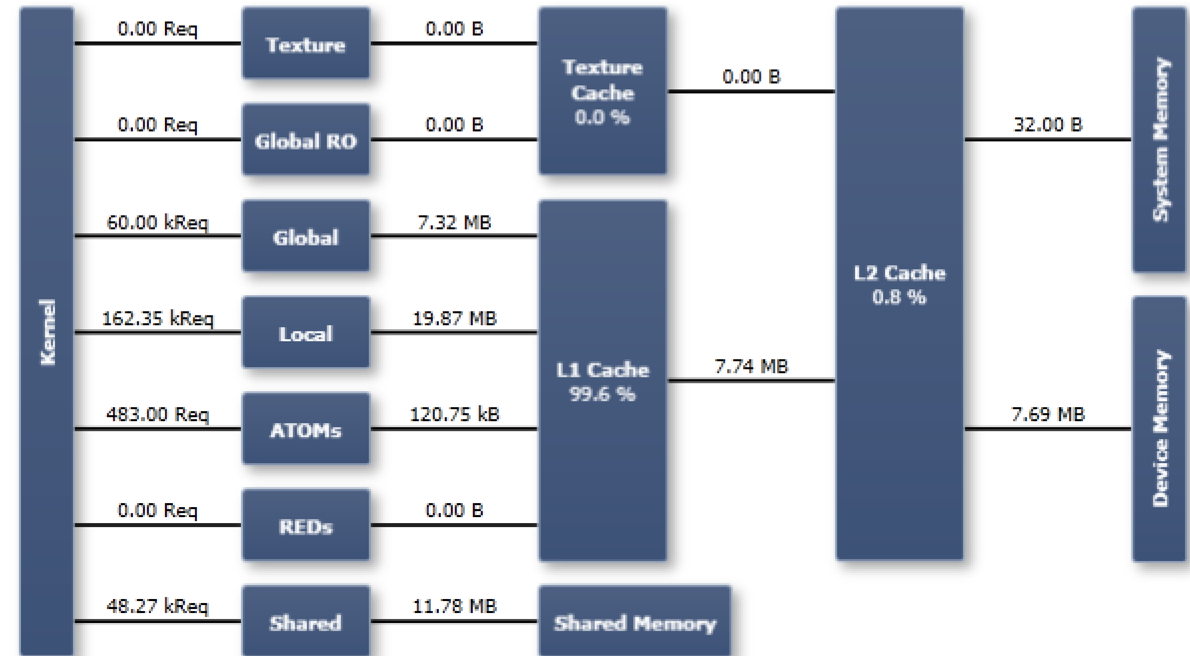


KIND OF MEMORY - GPU

- GPU is split in computation units
- Each of them is very similar to the CPU architecture
- But in GPU we have more kind of memories!
  - Each with their own tradeoffs



KIND OF MEMORY - GPU

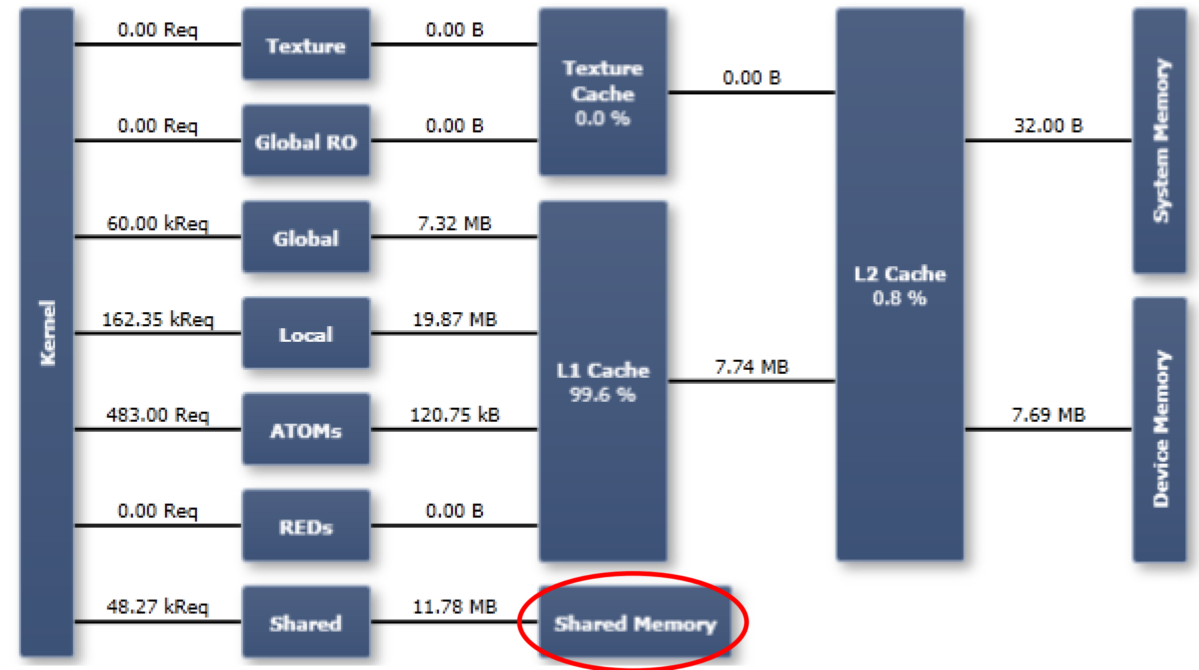


- GPU uses hierarchies of memory
  - The closer the memory is to the thread:
    - the faster it is
    - the smaller it is
  - It is costly/SLOW to communicate between different memories
    - ⇒ slow to communicate between threads!
    - ⇒ slower to communicate between blocks
    - ⇒ slowerer to communicate between multiprocessors
    - ⇒ slowererer to communicate between the VRAM (GPU) and the RAM (CPU)
  - No automatic control of race conditions at the memory layer
    - You need to check for race conditions yourself in code!
- With GPUs: You are in CONTROL on which variable goes in which memory!

SHARED MEMORY

- SHARED memory is:
  - very fast
  - Accessible by block
- You need to:
  - declare a shared array in the kernel, or
  - call the kernel with the corresponding size in C++

```
#define MEM_SIZE 32
__global__ void kernel(const int* in, int* out) {
    // Static allocation
    __shared__ int shared_array[MEM_SIZE];

    // Useful code which exploit the shared memory
}
__global__ void kernelDyn(const int* in, int* out) {
    // Dynamic allocation
    extern __shared__ int shared_array[];
}
int main () {
    // [...]
    kernel<<<block_size, threads_per_block>>>(in, out);
    // [...]
    kernelDyn<<<block_size, threads_per_block,
                MEM_SIZE*sizeof(int)>>>(in, out);
    // [...]
}
```

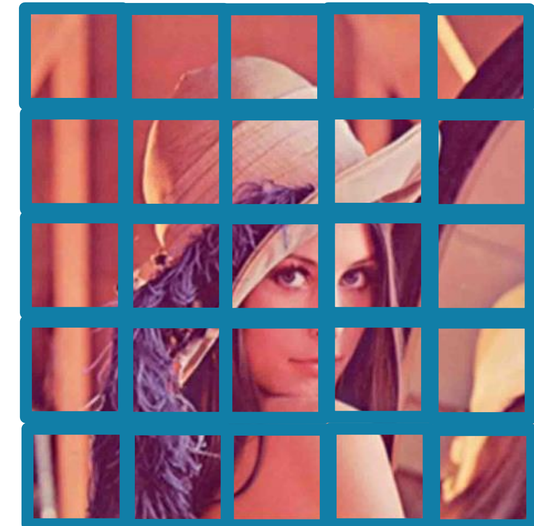


SHARED MEMORY

```
Device name: NVIDIA GeForce RTX 3090
Compute Capability: 8.6
Shared Memory per Block: 49152 bytes
Shared Memory per Multiprocessor: 102400 bytes
```

- Shared memory is fast but small
  - You can check the amount of shared memory available per block using cudaGetDeviceProperties()
- FullHD image = 6,220,800 elements → 6,220,800 bytes
- Whole image does not fit in the shared memory
- Each block is going to process a part of the image
- For each block, put the part of image needed to process this block in the shared memory

Lena



SOME (PARTS OF) KERNELS ARE NOT PARALLELIZABLE

- Sometime you NEED synchronization between threads
- Eg:
  - Every thread initialize the shared memory before doing something
  - Synchronize them to be sure the memory in initialized
  - Do some other operation
- Command:
  - __syncthreads();
  - Use it only if necessary! (race conditions)

```
__global__ void func(int *arr, int *out)
{
    __shared__ int local_array[THREADS_PER_BLOCK];
    int idx = blockIdx.x * blockDim.x+ threadIdx.x;

    // Initialize the data in local memory
    local_array[threadIdx.x] = arr[idx];

    // Synchronize the local threads per block
    // = Barrier, wait they are all done.
    __syncthreads();

    out[idx] = ops_on_local_array(threadIdx.x);
}
```

SOME KERNELS ARE NOT PARALLELIZABLE

- Sometime you NEED synchronization between **blocks**
- Eg:
  - We want to dot product two vectors with 100.000+ elements
  - We divide the vectors in several blocks
  - We need to add the results of each block together
- Can't synchronize between block
  - Blocks run in random order
  - Can't wait until all the block are done with an operation
- But we can make sure blocks don't access the same resource at the same time with command:
  - atomicAdd(out, variable); (addition between blocks)
  - Use it only if necessary! (race conditions)
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions
    - atomicAdd, atomicSub, atomicMin, atomicMax, atomicCAS (compare), atomicAnd, atomicOr, atomicXor, etc.

$$a = (a_0, a_1, \cdots, a_T, a_{T+1}, \cdots, a_{2T}, \cdots, a_N)$$
$$b = (b_0, b_1, \cdots, b_T, b_{T+1}, \cdots, b_{2T}, \cdots, b_N)$$

$$a.b = \sum_{i=1}^{N} a_i b^i$$

$$a.b = \sum_{i=1}^{N/T} \sum_{j=iN}^{T} a_i b^i$$

N blocks of T threads

BASIC APPLICATION: DOT PRODUCT

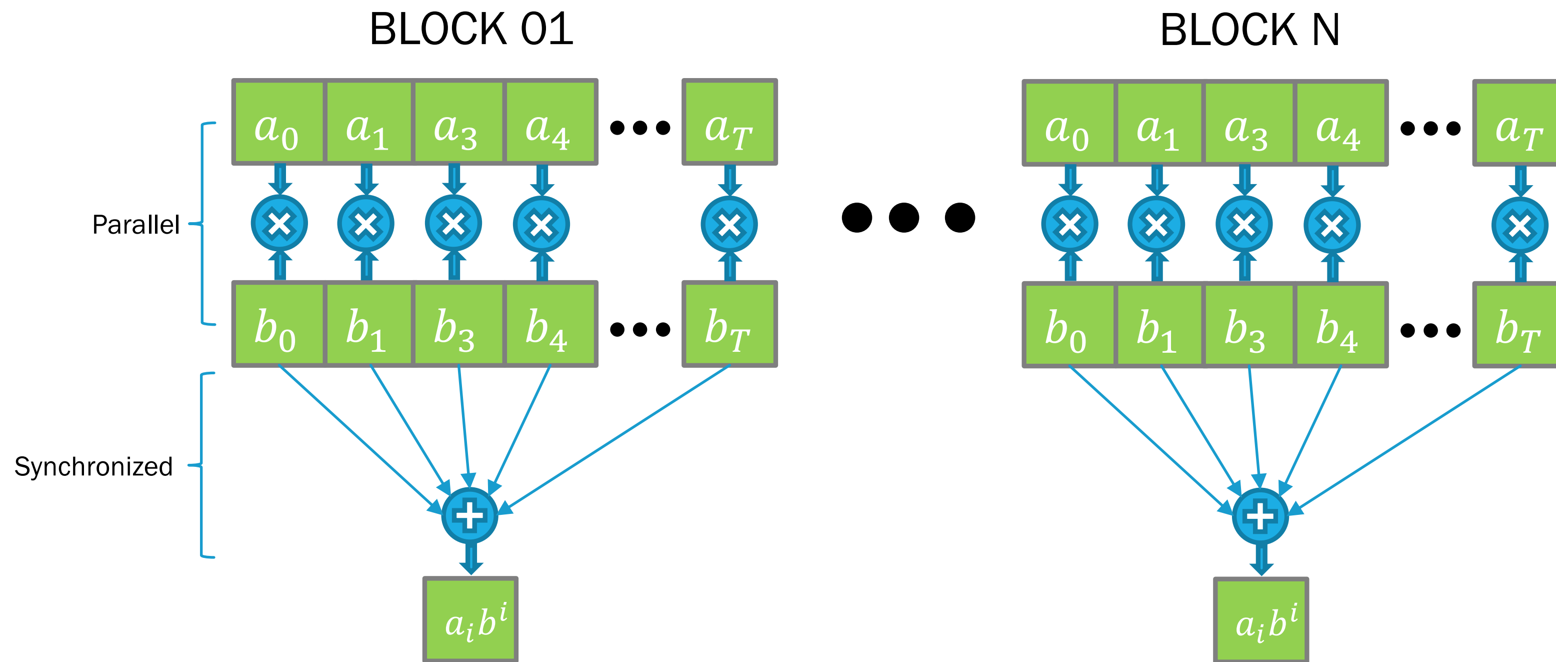
- We want to dot product two vectors with 100.000+ elements
- N blocks of T threads
- We divide the vectors in several blocks
- We need to add the results of each block together in a synchronized way

$$a = (a_0, a_1, \cdots, a_T, a_{T+1}, \cdots, a_{2T}, \cdots, a_N)$$
$$b = (b_0, b_1, \cdots, b_T, b_{T+1}, \cdots, b_{2T}, \cdots, b_N)$$

$$a.b = \sum_{i=1}^{N} a_i b^i$$

$$a.b = \sum_{i=1}^{N/T} \sum_{j=iN}^{T} a_i b^i$$

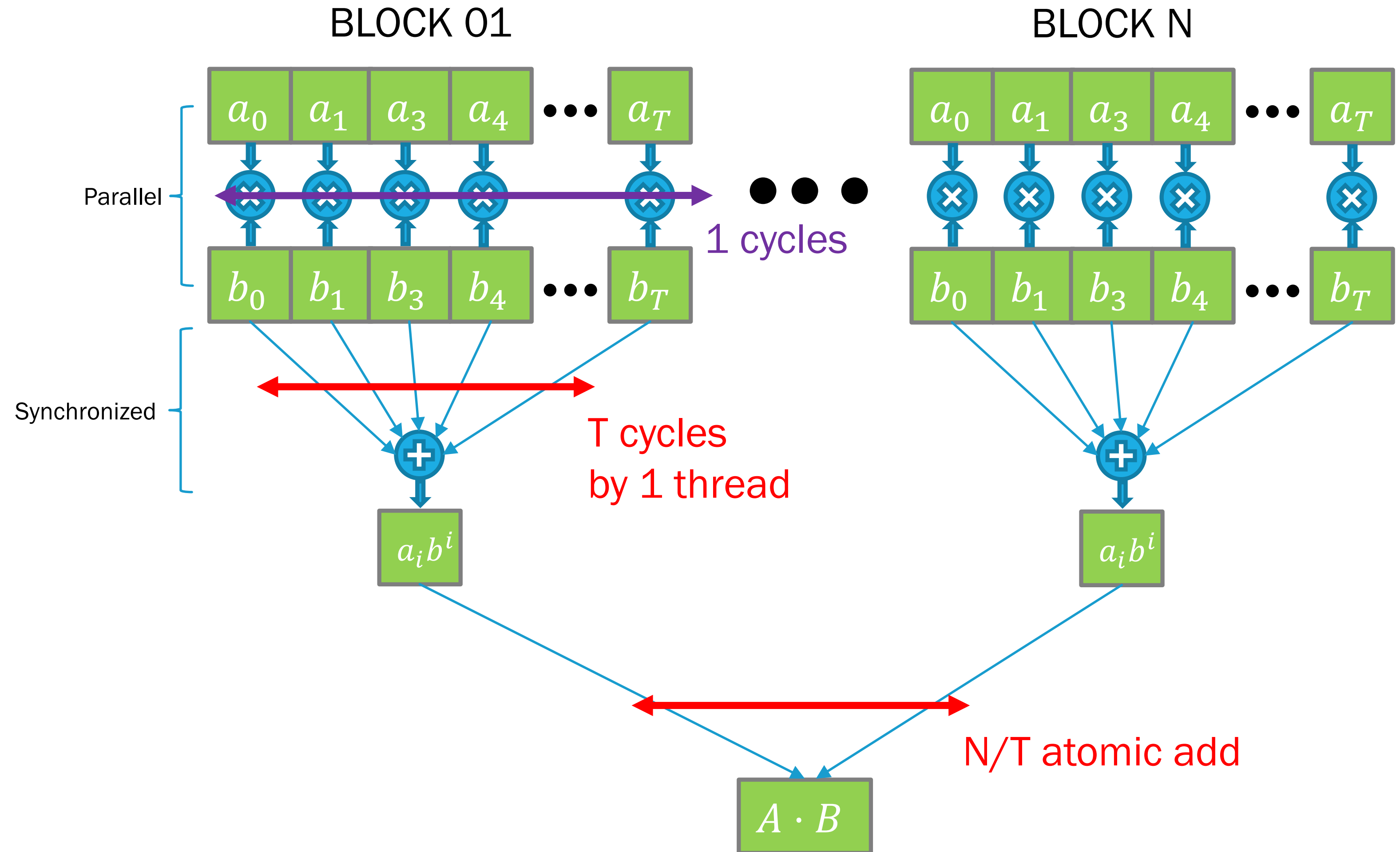
```
__global__ void dot(int* a, int* b, int* out)
{
    __shared__ int local_array[THREADS_PER_BLOCK];
    ind idx = blockIdx.x * blockDim.x + threadIdx.x;

    local_array[threadIdx.x] = a[idx] * b[idx];

    // We wait all the threads are done per block
    __syncthreads();

    // Thread 0 for each block handle the sum!
    if (threadIdx.x == 0) {
        int sum = 0;
        for (int i = 0; i < THREADS_PER_BLOCK; ++i)
        {
            sum += local_array[i];
        }
        // We add/synchronize the blocks!
        atomicAdd(out, sum);
    }
}
```



Remark: the “if” creates a warp divergence!

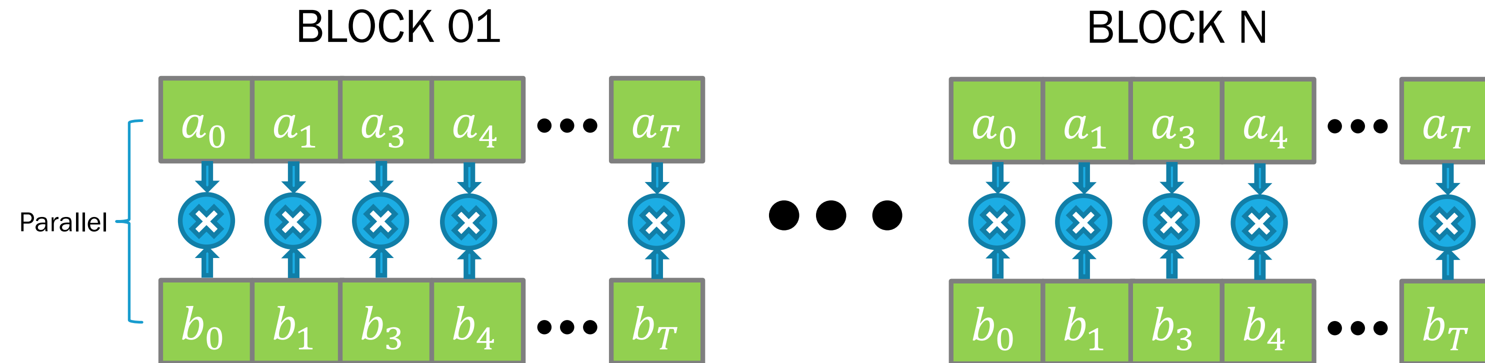
BASIC APPLICATION: DOT PRODUCT – DYADIC SUM

- Operations:
  - Each block performs:
    - 1 cycle multiplication
    - T cycles addition (synchronized)
  - Between blocks:
    - N/T atomic additions
- Can we do better?



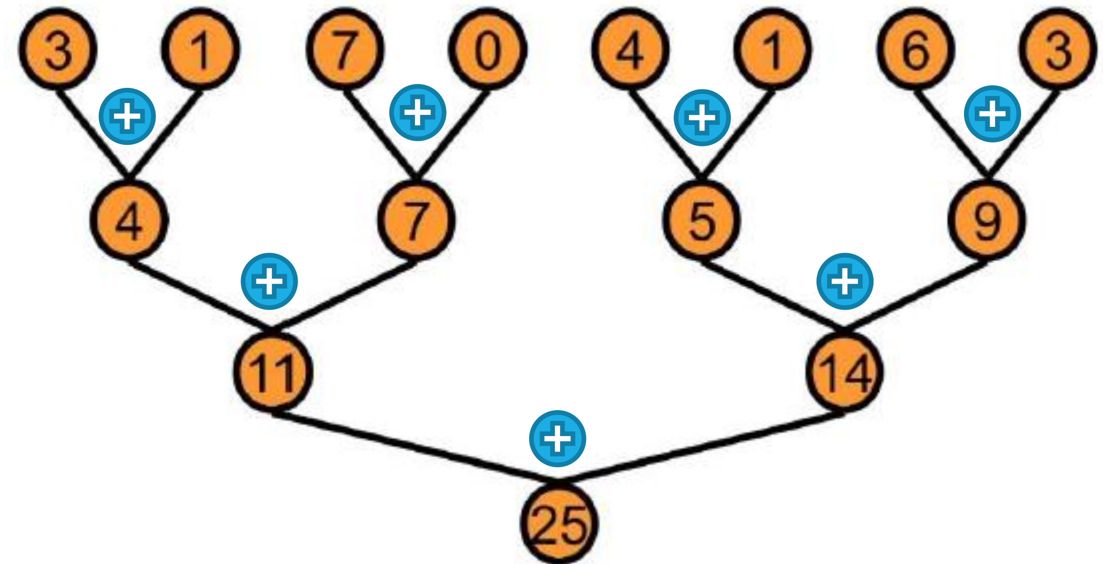
BASIC APPLICATION: DOT PRODUCT – DYADIC SUM

- The dot product is always:
  - Parallelizable products
  - Non parallelized sums



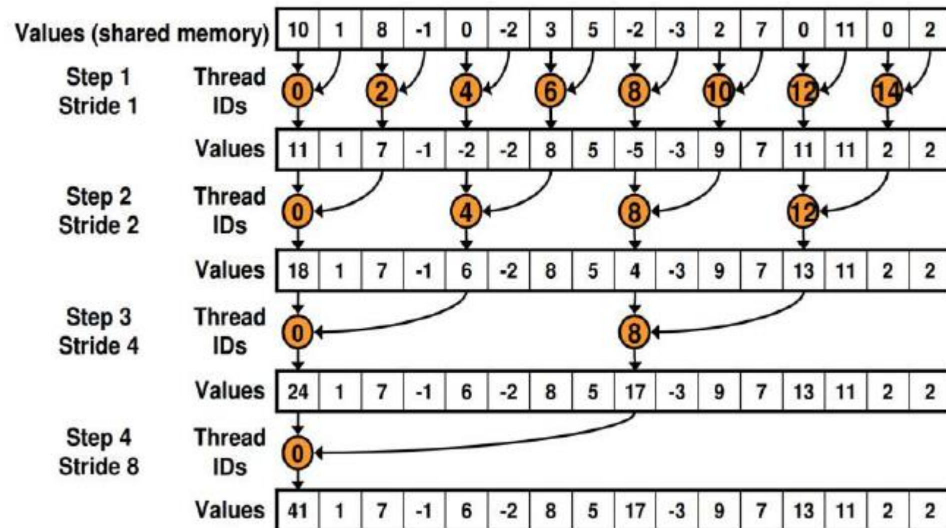
- What if we have a separate kernel to perform the sums?

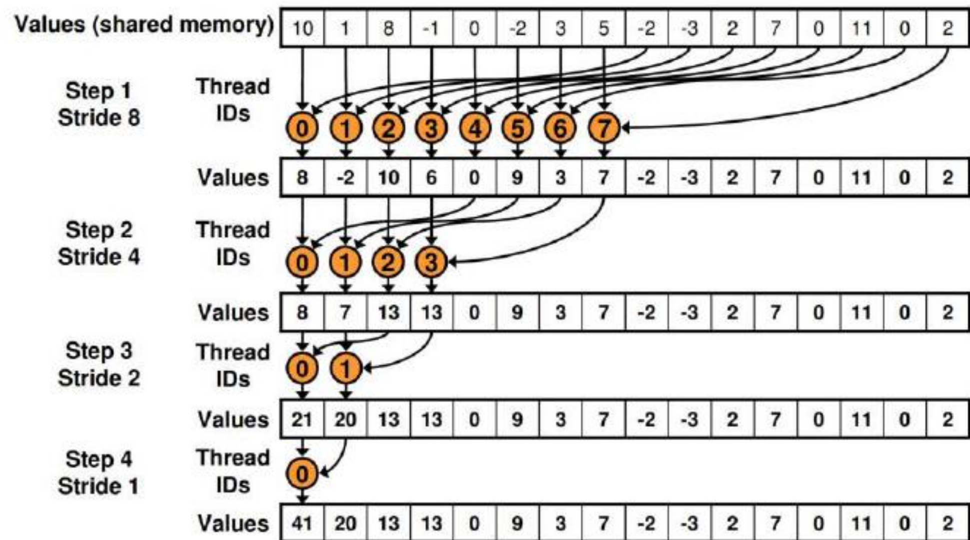
- Main idea we can exploit for the sums:
  - + is commutative and associative in $(\mathbb{R}, +, \times)$
  - = We can perform additions in any order



BASIC APPLICATION: DOT PRODUCT – DYADIC SUM

- Remarks:
  - We want to exploit shared memory (avoid global access)
    - Use “local_array” of shared memory which already contains all the multiplications per block
  - We can think of several strategies to perform the sum:

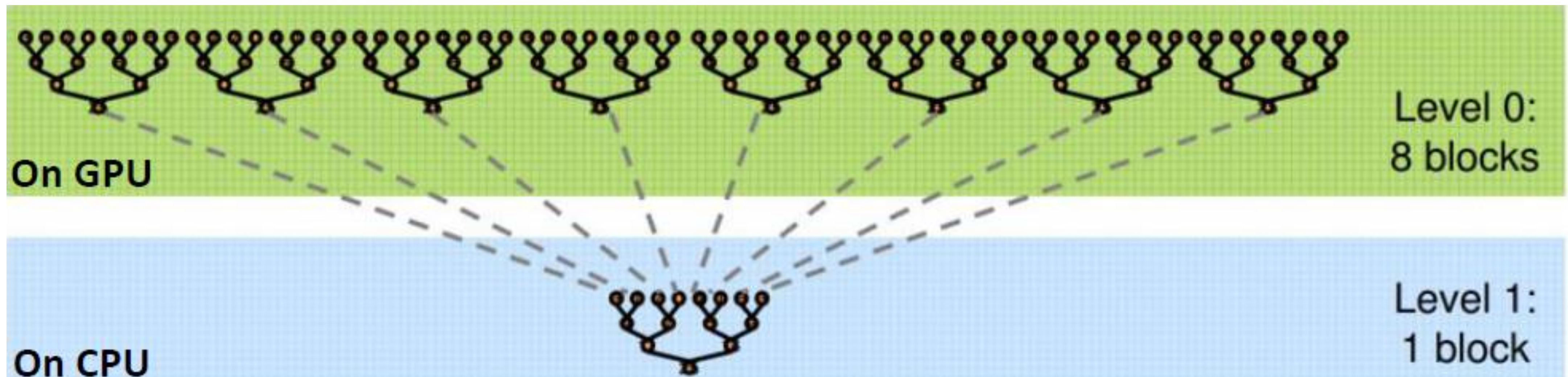




Profile them to know which one is the best and why!

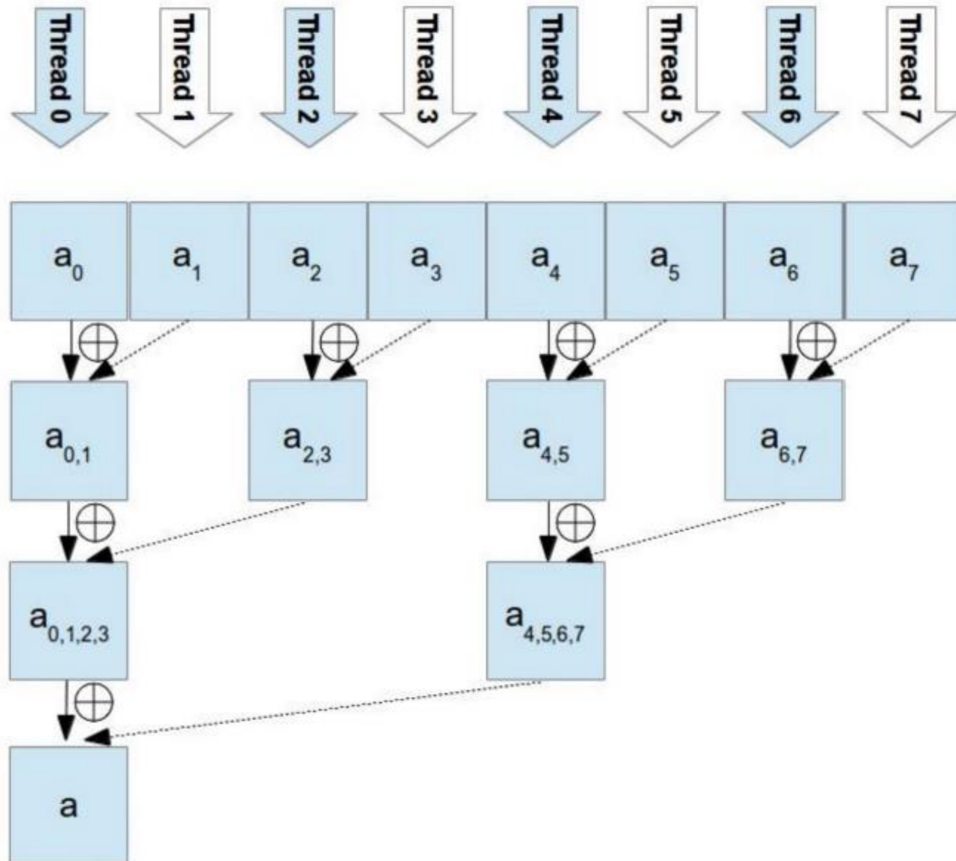
BASIC APPLICATION: DOT PRODUCT – DYADIC SUM

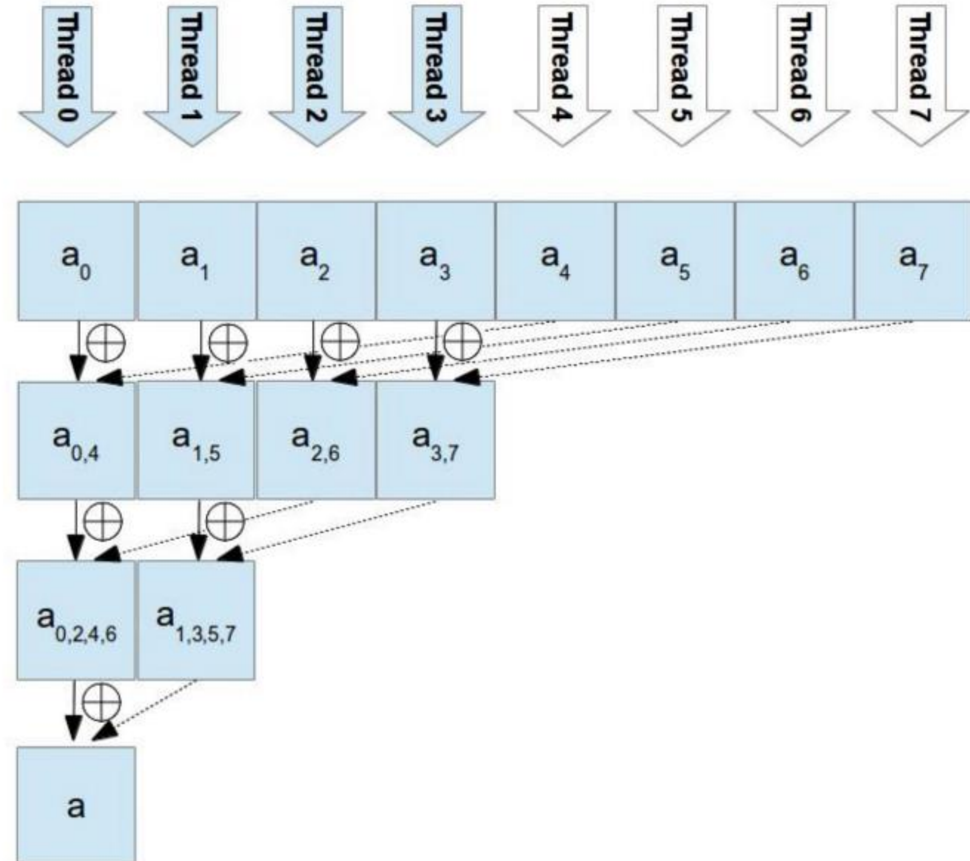
- In both cases, the GPU can be overkill to perform the additions
  - At each loop, the number of working threads is divided by 2
- The last steps can be performed on the CPU
  - To avoid running kernels with very few active threads
  - The context change is more important here than doing CPU operations



EXERCISE: DOT PRODUCT – DYADIC SUM

- Implement the dot product
- The dot product with the two strategies of the dyadic sum
  - Analyze with the profiler
  - Based on the results of the profiler, you can decide at which number of elements you should run the sums on the CPU

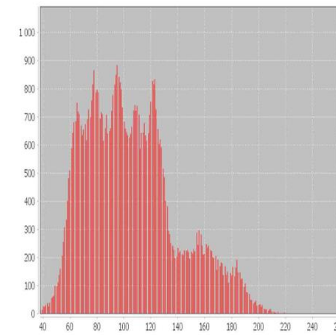
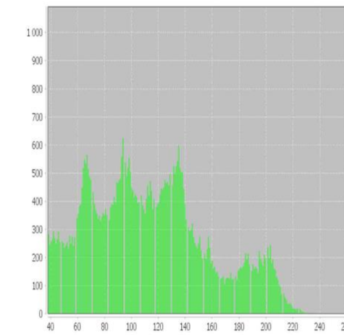
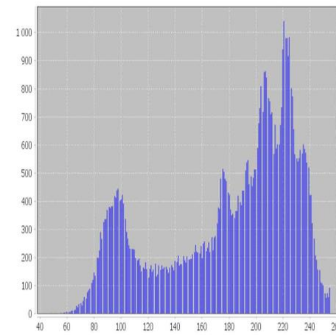




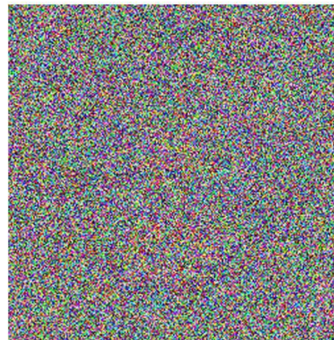
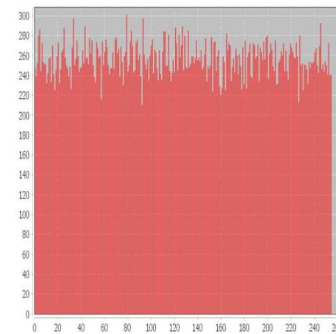
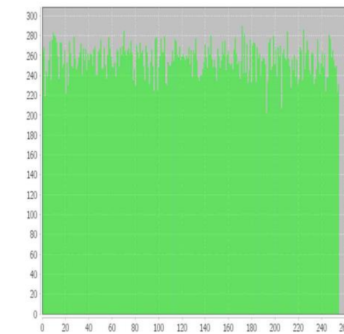
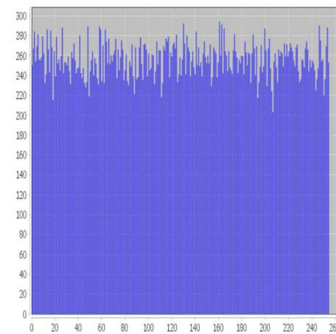
BASIC APPLICATION: HISTOGRAM

- Computing histograms is a very usual task
- Used in image processing
  - Mapping
  - Color corrections
  - Noise removal
  - Etc.
- Each "bin" represent one value for a color
  - 256 bin in image processing applications
- We count the number of occurrences of each "bin" in the image
- We need to compute them fast!
  - Some bin will be used more than others
  - Race conditions → We need atomic additions

Lena

Random noise*

*: Actually this comes from a paper on cryptography of the same image
A new image encryption algorithm using random numbers generation of two matrices and bit-shift operators

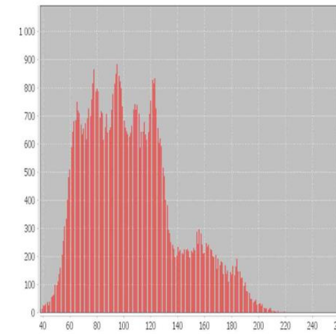
BASIC APPLICATION: HISTOGRAM

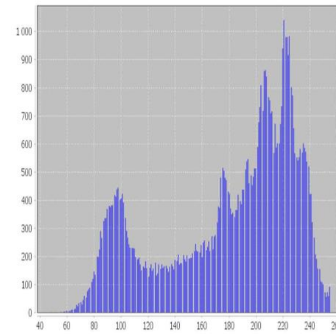
- In C++

```
Image img('file'); // grayscale
int histogram[255] = {0}
for (int x = 0; x < img.width; ++x) {
    for (int y = 0; y < img.height; ++y) {
        int value = img[x][y];
        histogram[value] += 1;
    }
}
```

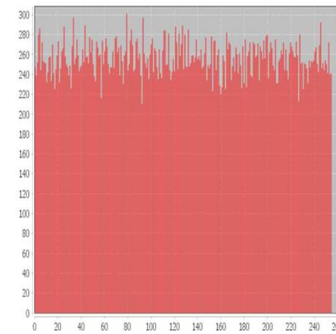
- To load images:
  - OpenCV (difficult to include in Visual Studio)
    - https://opencv.org/
  - stb_image.h (header only library)
    - https://github.com/nothings/stb/blob/master/stb_image.h

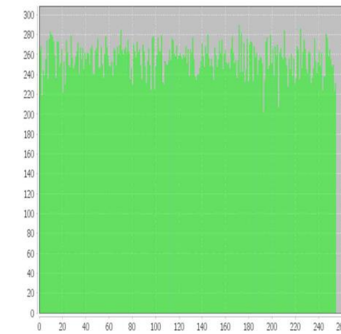
Lena

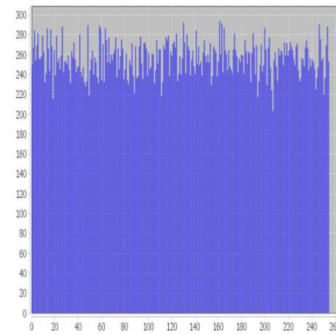




Random noise*







*: Actually this comes from a paper on cryptography of the same image
A new image encryption algorithm using random numbers generation of two matrices and bit-shift operators

BASIC APPLICATION: HISTOGRAM

```
Image img('file'); // grayscale
int histogram[255] = {0}
for (int x = 0; x < img.width; ++x) {
    for (int y = 0; y < img.height; ++y) {
        int value = img[x][y];
        histogram[value] += 1;
    }
}
```

```
// [...]
int dataSize = img.width * img.height;
int N_threads = 1024;
dim3 block_size(dataSize ( + (N_threads-1))/N_threads);
dim3 thread_size(N_threads);
histogram<<<block_size, thread_size>>>(dev_img, dev_hist, img.size);
// [...]
```

```
__global__ void histogram(unsigned char *img,
                          unsigned int * hist,
                          long N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= N)
        return;

    unsigned int color = img[idx];
    // Add 1 to the global memory histogram
    atomicAdd(&(hist[color]), 1);
}
```

- How to parallelize this?
  - Some bin will be used more than others
  - Race conditions → We need atomic additions
    - In global memory
    - But also in shared memory!
  - Challenge: output location for each element is not known prior to reading its value
- Idea 1:
  - Lunch as many threads as the image size

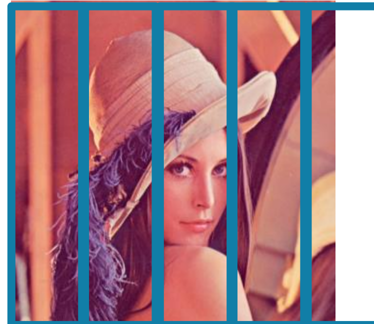
- Advantages
  - Very similar to CPU implementation
- Drawbacks
  - Very slow
    - Access to the global memory
    - AtomicAdds

BASIC APPLICATION: HISTOGRAM

```
// [...]
int dataSize = img.width * img.height;
int N_threads = 1024;
dim3 block_size(dataSize ( + (N_threads-1))/N_threads);
dim3 thread_size(N_threads);
histogram<<<block_size, thread_size>>>(dev_img, dev_hist, img.size);
// [...]
__global__ void histogram(unsigned char *img,
                          unsigned int * hist,
                          long N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= N)
        return;

    unsigned int color = img[idx];
    // Add 1 to the global memory histogram
    atomicAdd(&(hist[color]), 1);
}
```

```
// [...]
int dataSize = img.width * img.height;
int N_threads = 256;
dim3 block_size((img.width + (N_threads-1))/N_threads, 1, 1);
dim3 thread_size(N_threads);
histogram<<<block_size, thread_size>>>(dev_img, dev_hist, img.size);
// [...]
__global__ void histogram(unsigned char *img,
                          unsigned int * hist,
                          long N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= N)
        return;

    // offset for a row
    int width = blockDim.x * gridDim.x;

    // Each thread will work on a column of the image
    while (idx < N) {
        unsigned int color = img[idx];
        // Add 1 at the address idx IN global histo
        // We need atomicAdd to avoid race conditions
        atomicAdd( &hist[color], 1);

        // Each thread of a block -> a column
        idx += width;
    }
}
```

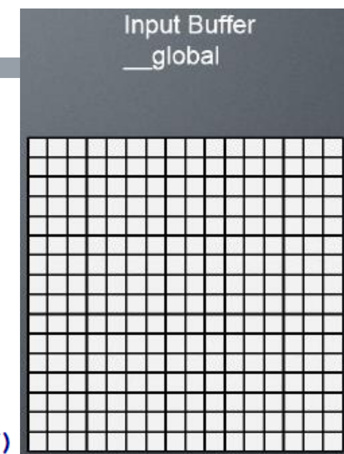
- Idea 2:
  - Executing a thread for only one pixel is overkill
  - Add a stride to make each thread work more
- Advantages
  - A little faster
- Drawbacks
  - Still slow



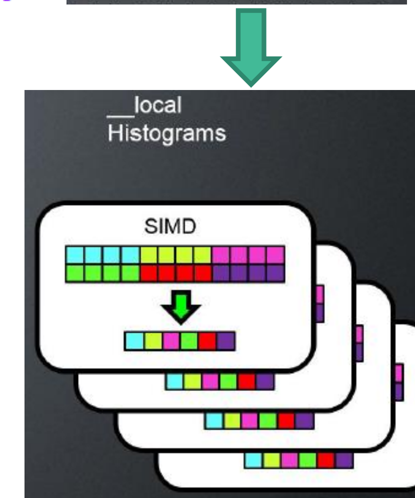
BASIC APPLICATION: HISTOGRAM

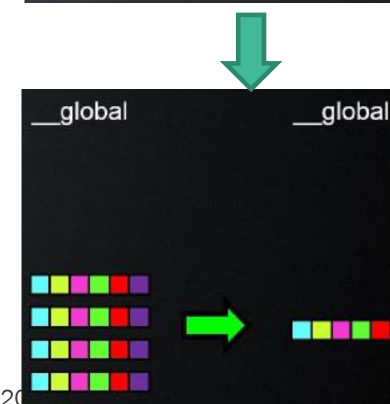
```
// [...]
int dataSize = img.width * img.height;
int N_threads = 256;
dim3 block_size((img.width + (N_threads-1))/N_threads, 1, 1);
dim3 thread_size(N_threads);
histogram<<<block_size, thread_size>>>(dev_img, dev_hist, img.size);
// [...]
__global__ void histogram(unsigned char *img,
                          unsigned int * hist,
                          long N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= N)
        return;

    // offset for a row
    int width = blockDim.x * gridDim.x;

    // Each thread will work on a column of the image
    while (idx < N) {
        unsigned int color = img[idx];
        // Add 1 at the address idx IN global histo
        // We need atomicAdd to avoid race conditions
        atomicAdd( &hist[color], 1);

        // Each thread of a block -> a column
        idx += width;
    }
}
```

- Idea 3:
  - Exploit the shared memory → Local histograms
- Advantages
  - Way faster
- Drawbacks
  - Yet not optimal ☺ https://developer.nvidia.com/blog/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/




```
// [...]
int dataSize = img.width * img.height;
int N_threads = 256; // histogram size!
dim3 block_size((img.width + (N_threads-1))/N_threads, 1, 1);
dim3 thread_size(N_threads);
size_t shared_size = 256 * sizeof(unsigned int); // local histogram !
histogram<<<block_size, thread_size, shared_size>>>(dev_img, dev_hist, img.size);
// [...]
__global__ void histogram(unsigned char *img, unsigned int * hist, long N)
{
    // Shared memory for local histogram
    __shared__ unsigned int local_hist[256];
    local_hist[threadIdx.x] = 0;

    __syncthreads();

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // offset for a row
    int width = blockDim.x * gridDim.x;

    // Each thread will work on a column of the image
    while (idx < N) {
        unsigned int color = img[idx];
        // Add 1 at the address idx IN local_hist
        // We need atomicAdd to avoid race conditions
        atomicAdd( &local_hist[color], 1);

        // Each thread of a block -> a column
        idx += width;
    }
    // Wait all the threads to finish their column
    __syncthreads();

    // Block level synchronization
    // Each thread copy its own value
    atomicAdd( &(hist[threadIdx.x]), local_hist[threadIdx.x] );
}
```





EXERCISE: HISTOGRAM

- Use stb_image or OpenCV to load an image in your code
  - Both libraries let you read/write images on disk to check your results
- Write the CPU version of the histogram
  - To check your histogram values, you can save a ".csv" file on disk
  - Plot the histogram with python and matplotlib
- Try to write without the slides the 3 histograms kernels
  - Profile them with the CUDA profilers (compare the different statistics in the profilers)
  - Look at the speedups, are they what you expected?
  - Try different "stride" for version 2 & 3

- This exercise is important to learn how to use/integrate OpenCV/stb_image for the next sessions!