

# GPU optimization techniques and tools

**A collection of ideas to maybe improve your GPU performance**

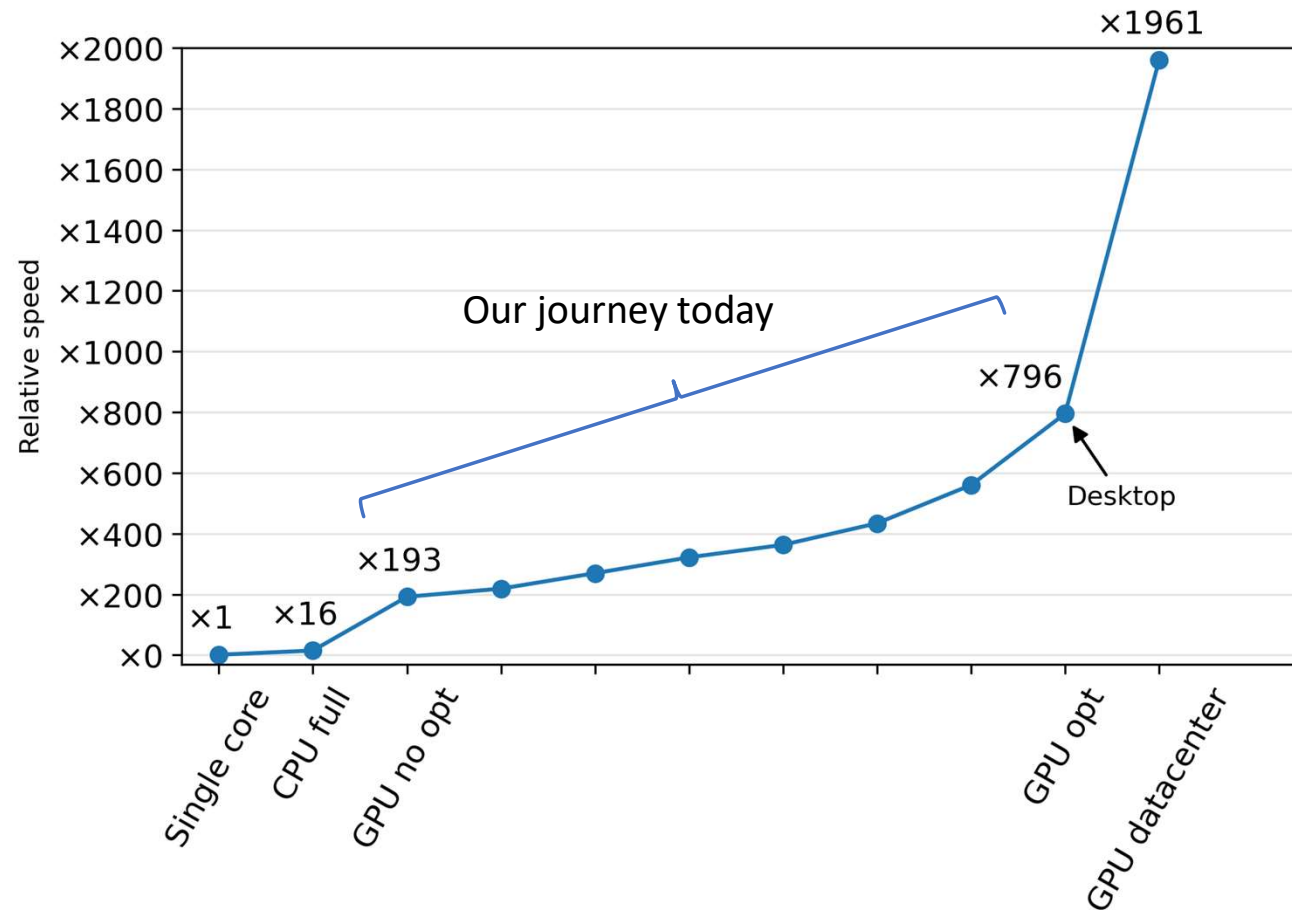
\*Presented data acquired over a two years period of hands-on experimentation and accumulated frustration and suffering.

Miguel De Le Court

UCLouvain, IMMC

# The importance of using GPU resources effectively

- Big gap between a naïve porting of a CPU-optimized code and a GPU-optimized code
- Writing efficient GPU code is HARD
- When it works : it's very fast



# Plan for this session

0. What is a GPU
1. double vs float
2. Locality
3. Coalescence
4. double literals
5. Occupancy limiters
6. Kernel fusion
7. Shared memory
8. Array of struct of array
9. Free compiler flags

# What is a GPU ?

Some differences VS a CPU include

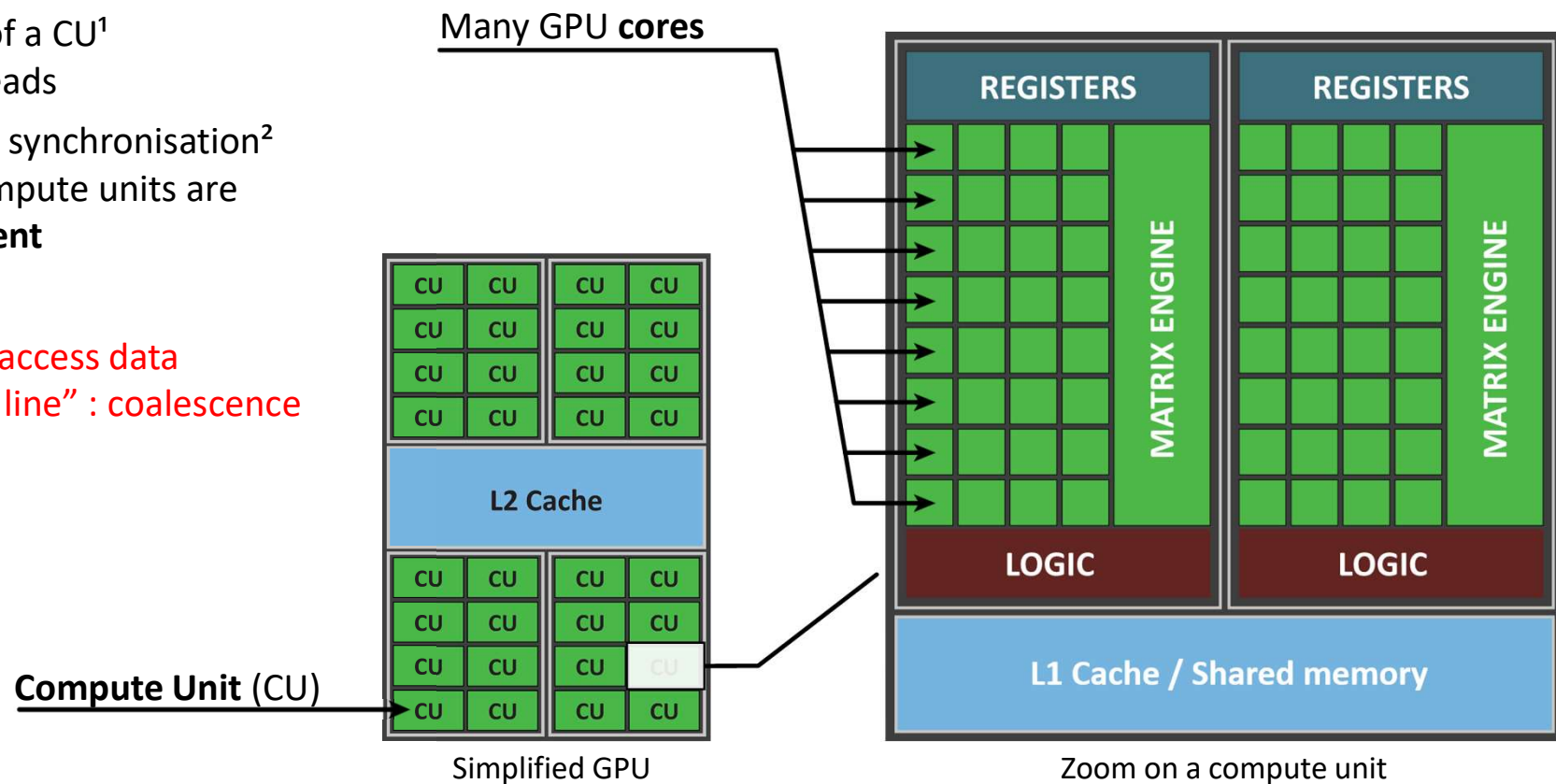
- SIMD-like execution model
- Coalescent memory access
- Very high memory latency
- Designed for higher arithmetic intensity
- Very limited cache per thread
- ...



# What is a GPU : the execution model

Defining characteristics:

- Cores of a CU<sup>1</sup> are **not independent**
- Computations inside of a CU<sup>1</sup> is the same for all threads
- No data exchange and synchronisation<sup>2</sup> outside of the CU. Compute units are **completely independent**
- Cores in a CU want to access data from the same “cache line” : coalescence



# Nomenclature

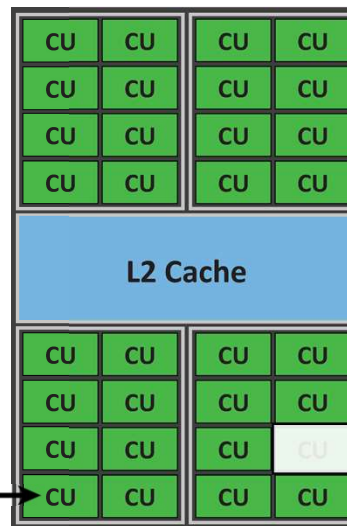
Green = NVIDIA

Blue = INTEL

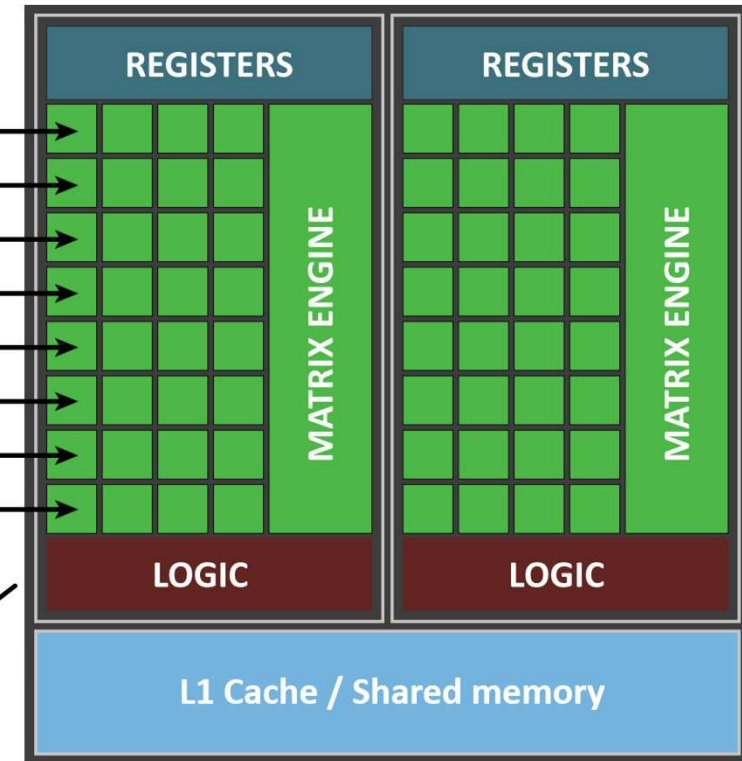
Red = AMD

(Cuda) cores = Shading Units = Stream processors

Streaming multiprocessor = Core = Compute Unit



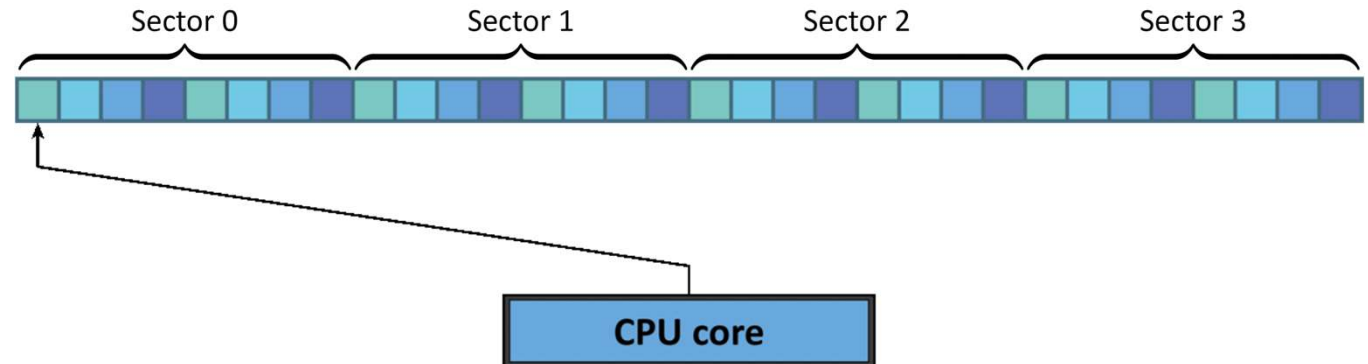
Simplified GPU



Zoom on a compute unit

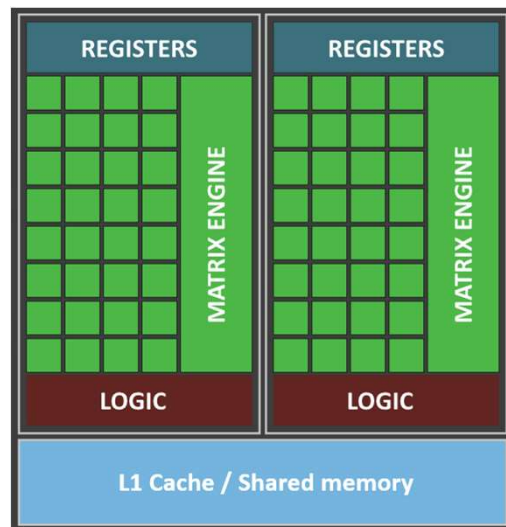
# Coalescent memory access

- On the CPU : we want to maximize locality

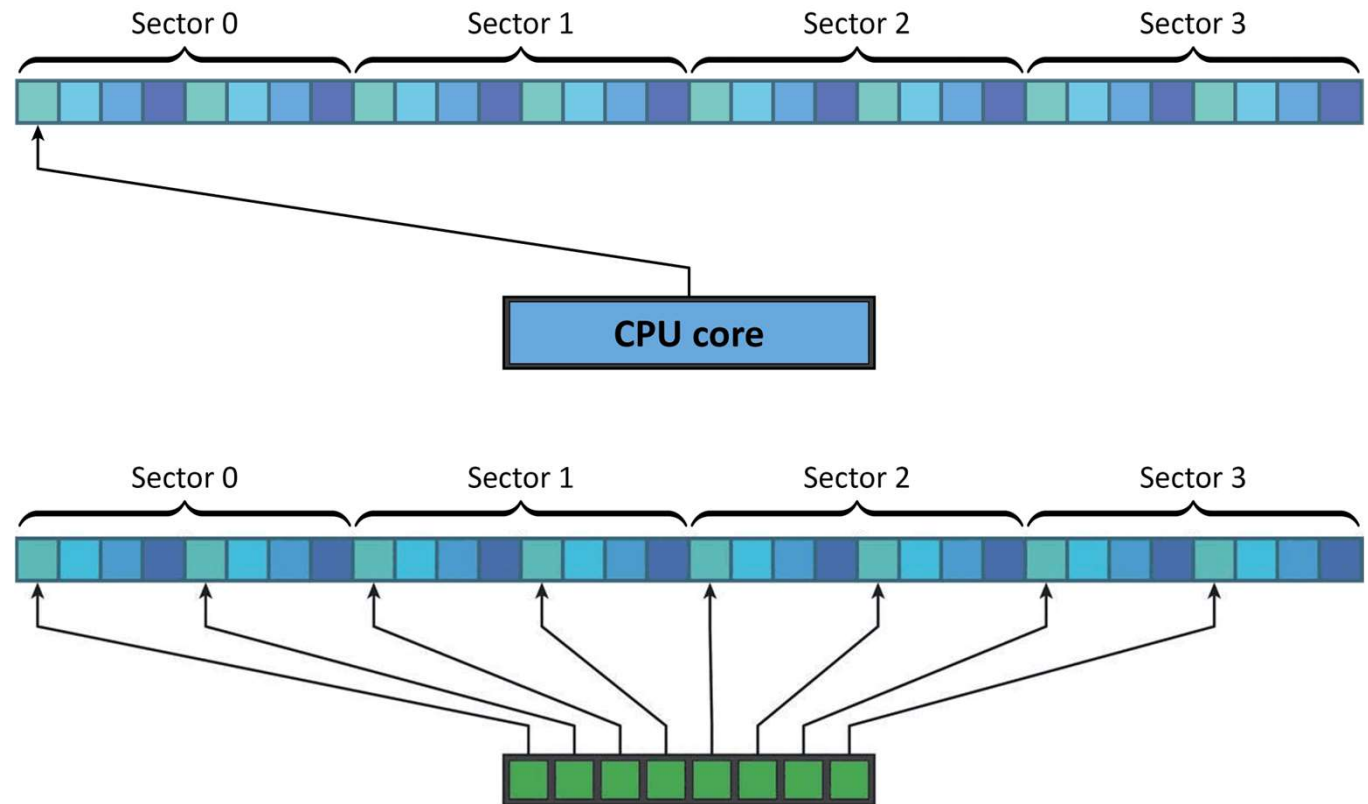


# Coalescent memory access

- On the CPU : we want to maximize locality
- On the GPU : data is accessed simultaneously
- Much smaller cache per core : data may not fit! → Excessive loads



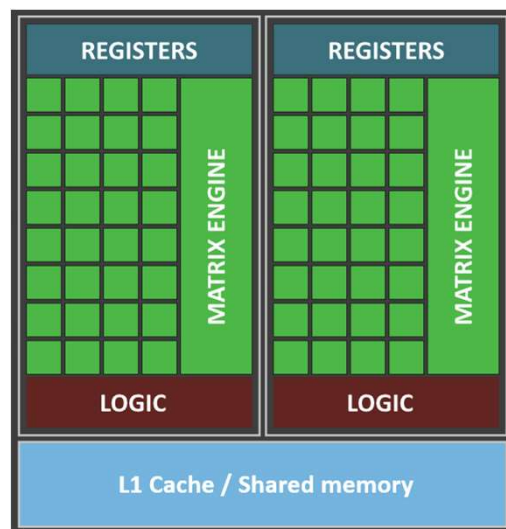
compute unit



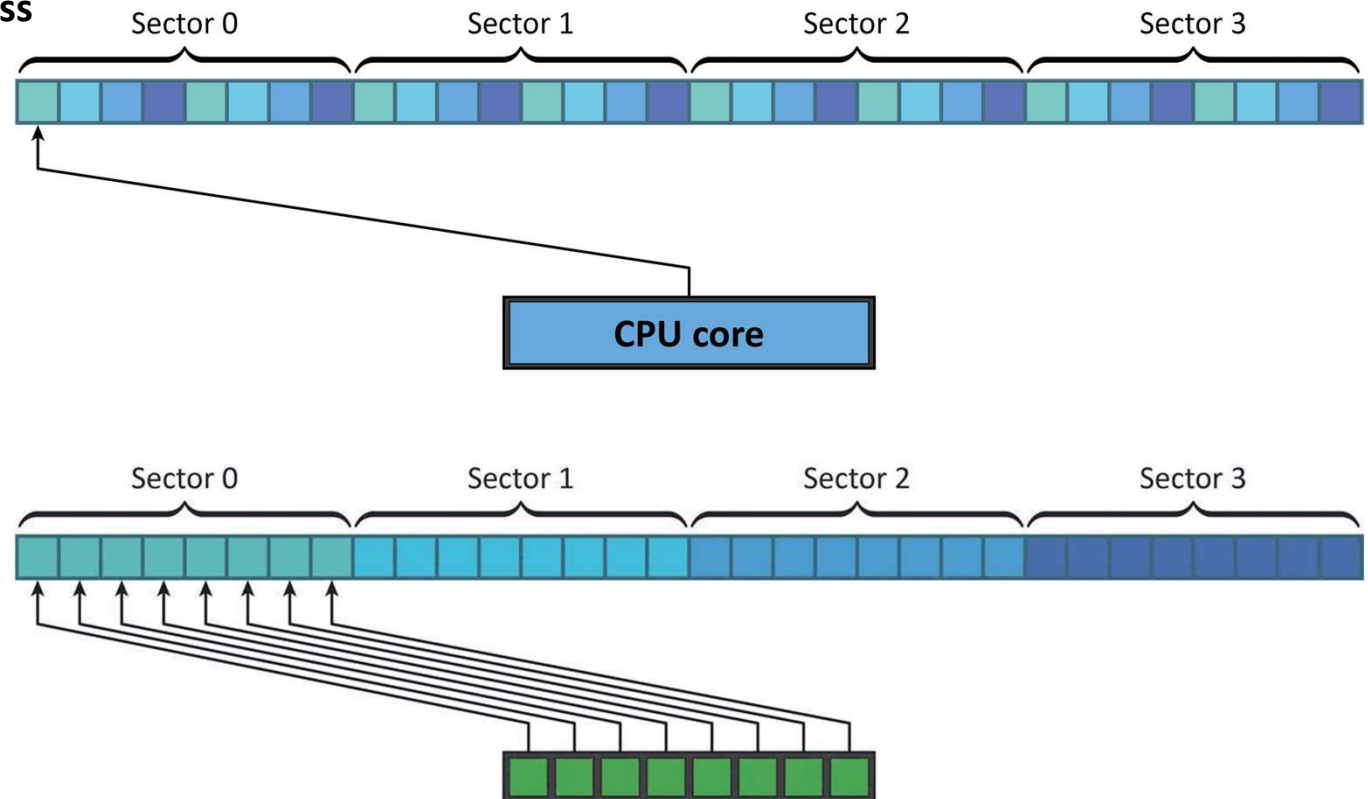


# Coalescent memory access

- On the CPU : we want to maximize locality
- On the GPU : data is accessed simultaneously
- Much smaller cache per core : data may not fit! → Excessive loads
- Optimal pattern : all cores from a CU read same sector : **One sector read per access**



compute unit



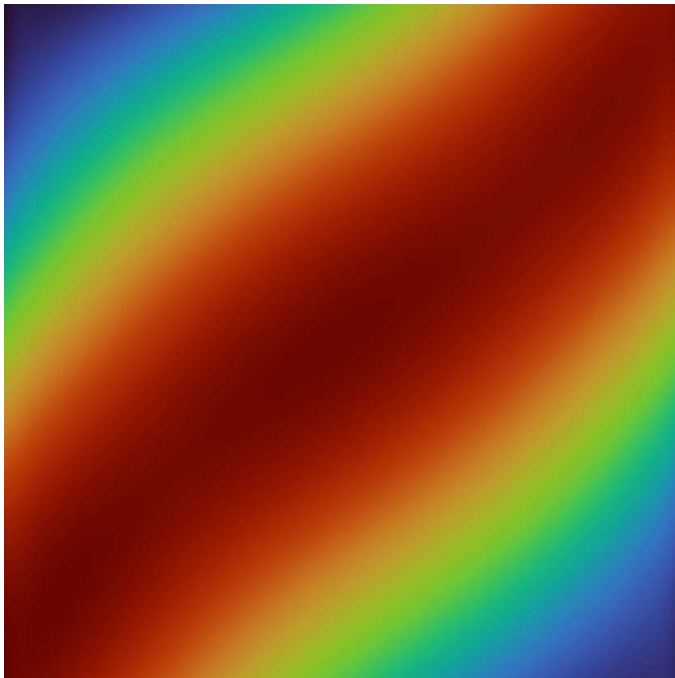
## 1 – Double vs float : which one should you choose, and why is it float\*?

- RTX3080:

- Float : 29.77 TFLOPS
- Double : 0.47 TFLOPS

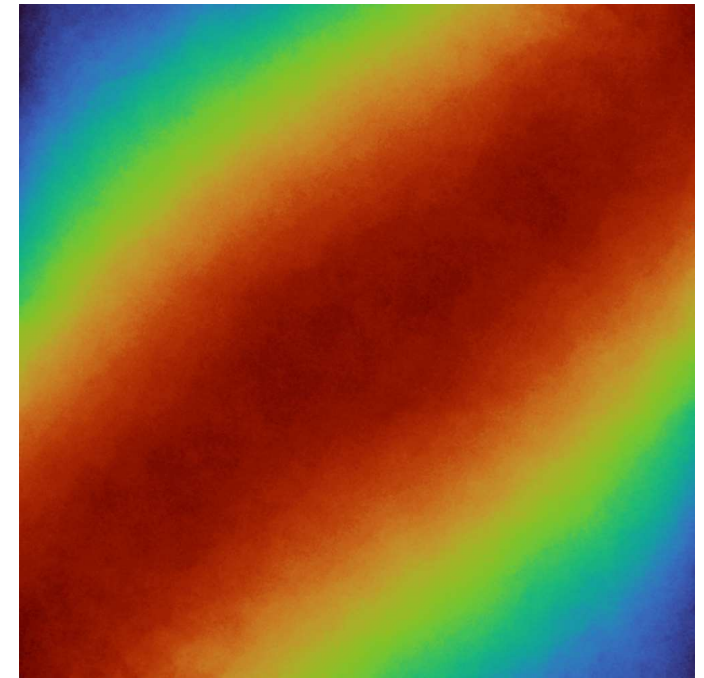
- A100:

- Float : 19.49 TFLOPS
- Double : 9.746 TFLOPS



Double

If you can, use floats



Float

## Where is the bottleneck ?

```
void dudt(const float* u, float* fu, ...){
```

```
    for(int i = 0; i < n; i++){
```

```
        // loading data
```

```
        float[4][3] local_u;
```

```
        local_u[...] = u[...];
```

} 4%

```
        // some heavy computation
```

Heavy processing

} 95%

```
        // writing back the result
```

```
        fu[...] = local_fu;
```

} 1%

```
    }
```

```
}
```

## Where is the bottleneck ?

```
__global__ void dudt_kernel(const float* u, float* fu, ...){  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    if (tid < n){  
        // loading data  
        float[4][3] local_u;  
        local_u[...] = u[...];  
  
        // some heavy computation  
  
        // writing back the result  
        fu[...] = local_fu;  
    }  
}
```

Heavy processing

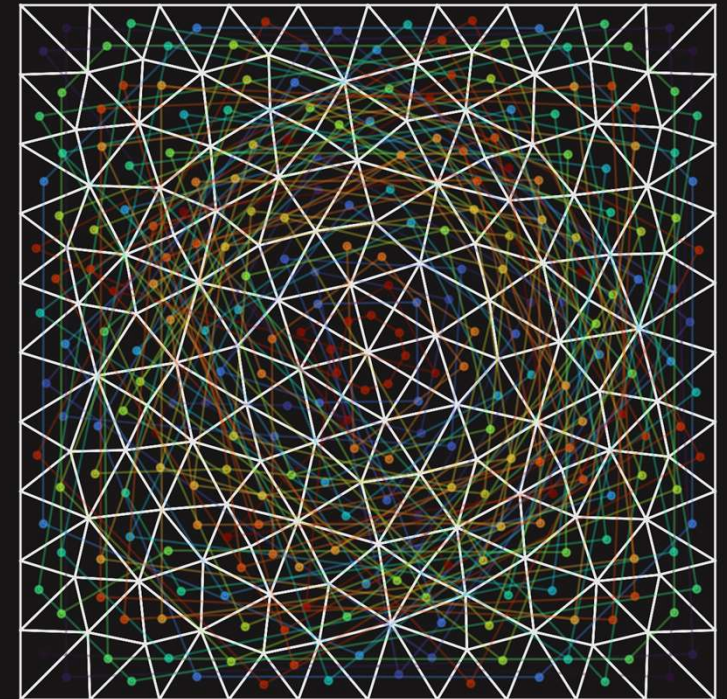
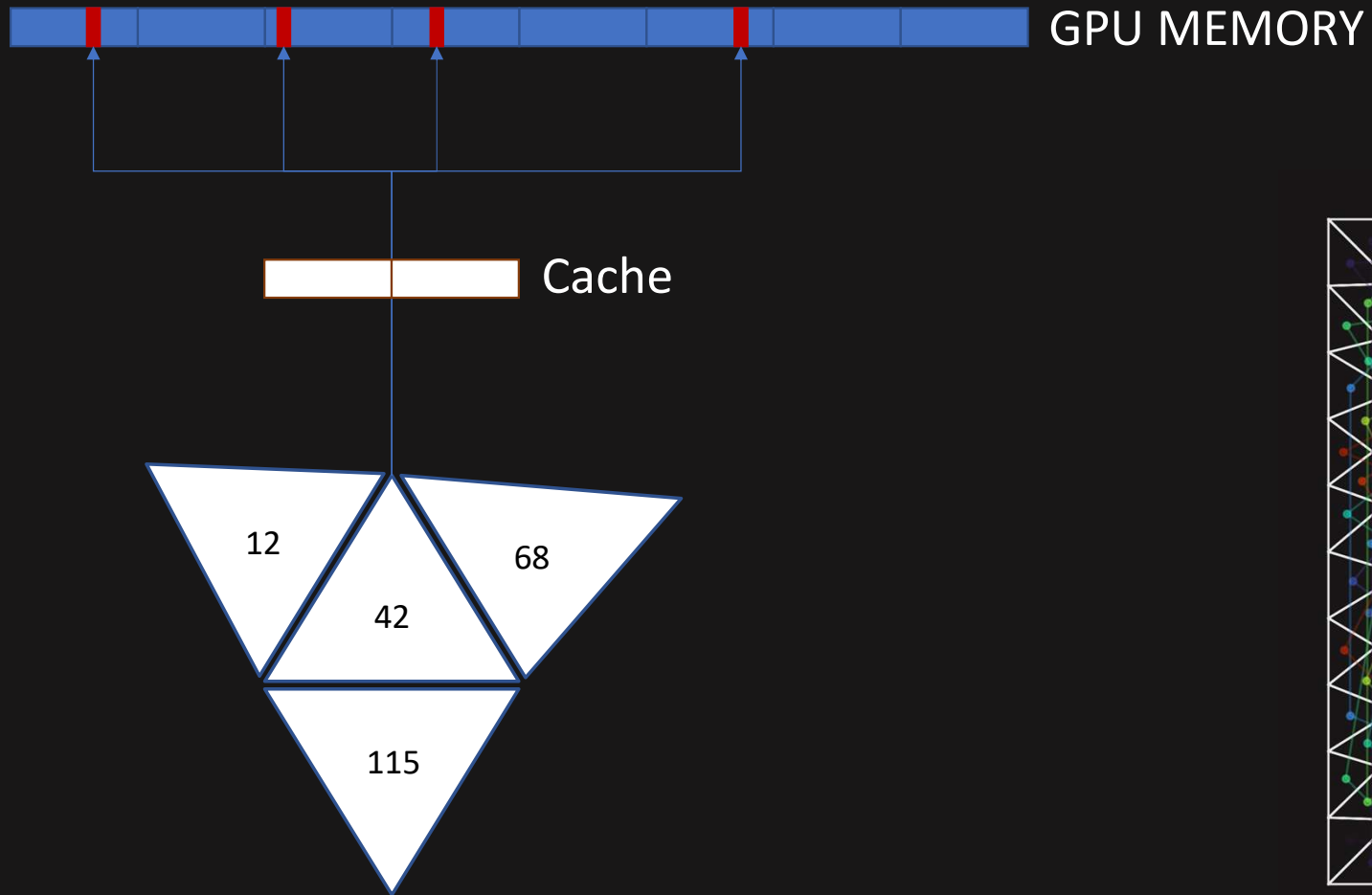
# Memory is (almost always) the bottleneck

```
__global__ void dudt_kernel(const float* u, float* fu, ...){  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    if (tid < n){  
        // loading data  
        float[4][3] local_u;  
        local_u[...] = u[...];  
  
        // some heavy computation  
  
        // writing back the result  
        fu[...] = local_fu;  
    }  
}
```

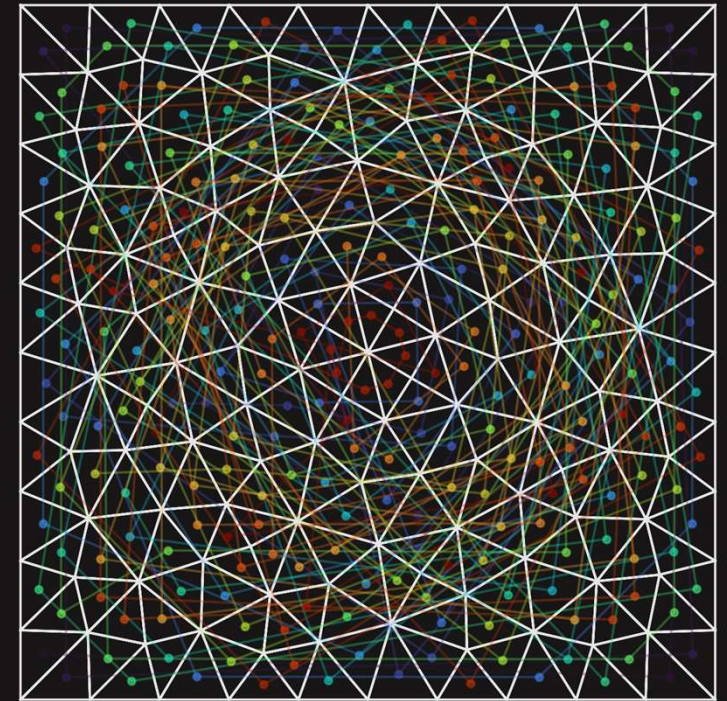
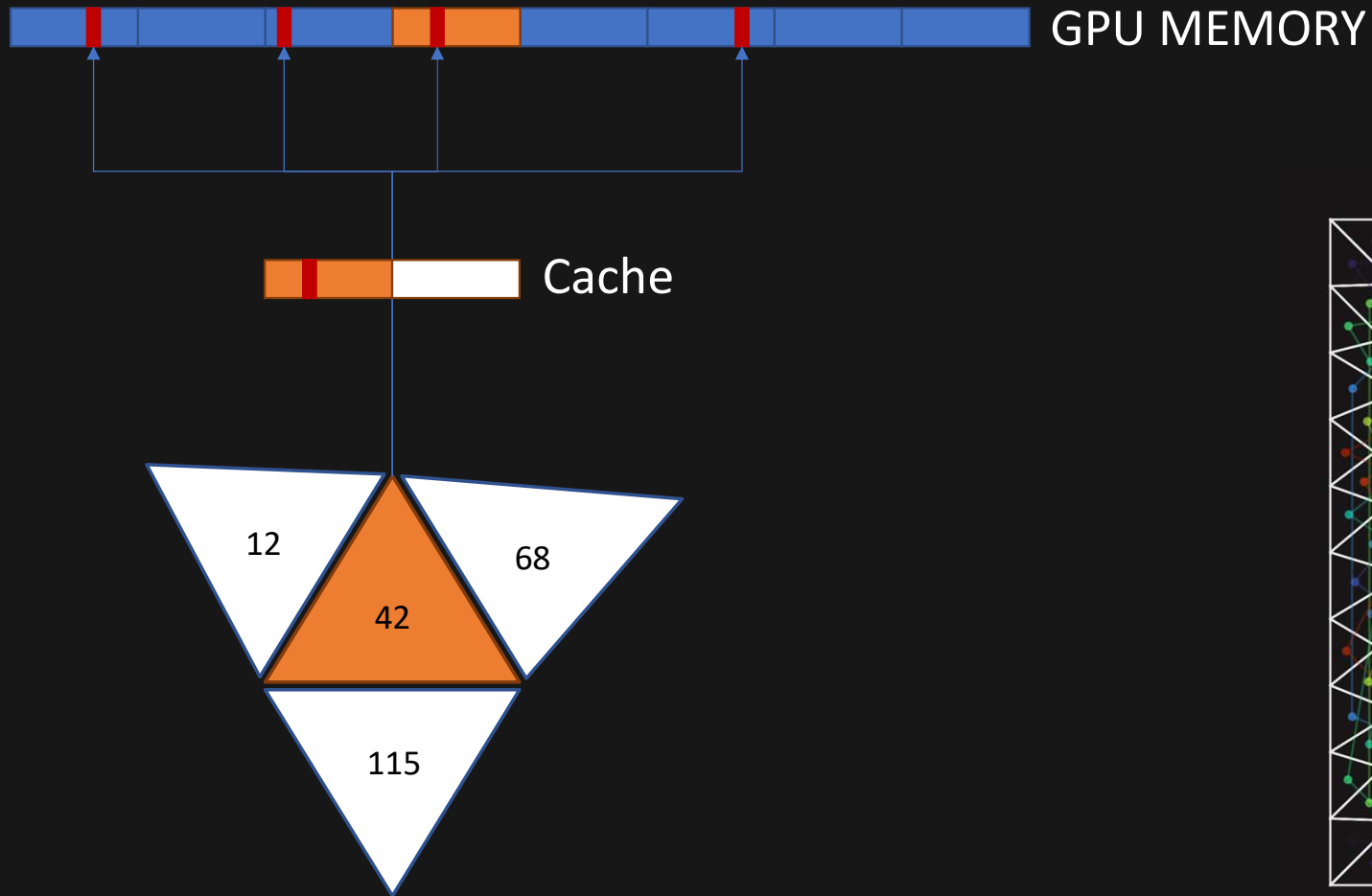
The diagram illustrates the execution flow of the kernel. It is divided into three main stages, each represented by a bracket on the right side of the code block:

- 55%**: This stage corresponds to the data loading phase, where the input data `u` is loaded into a local array `local_u`. The code lines for this stage are `float[4][3] local_u;` and `local_u[...] = u[...];`.
- 30%**: This stage corresponds to the heavy processing phase, which is represented by a large blue box labeled "Heavy processing". The code line for this stage is `// some heavy computation`.
- 15%**: This stage corresponds to the result writing phase, where the result is written back to the output array `fu`. The code lines for this stage are `// writing back the result` and `fu[...] = local_fu;`.

## 2 – Importance of memory locality

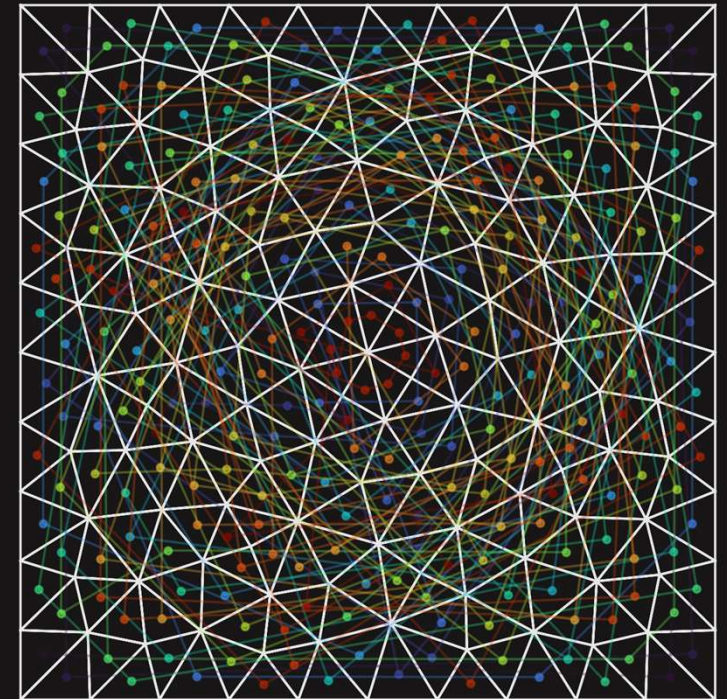
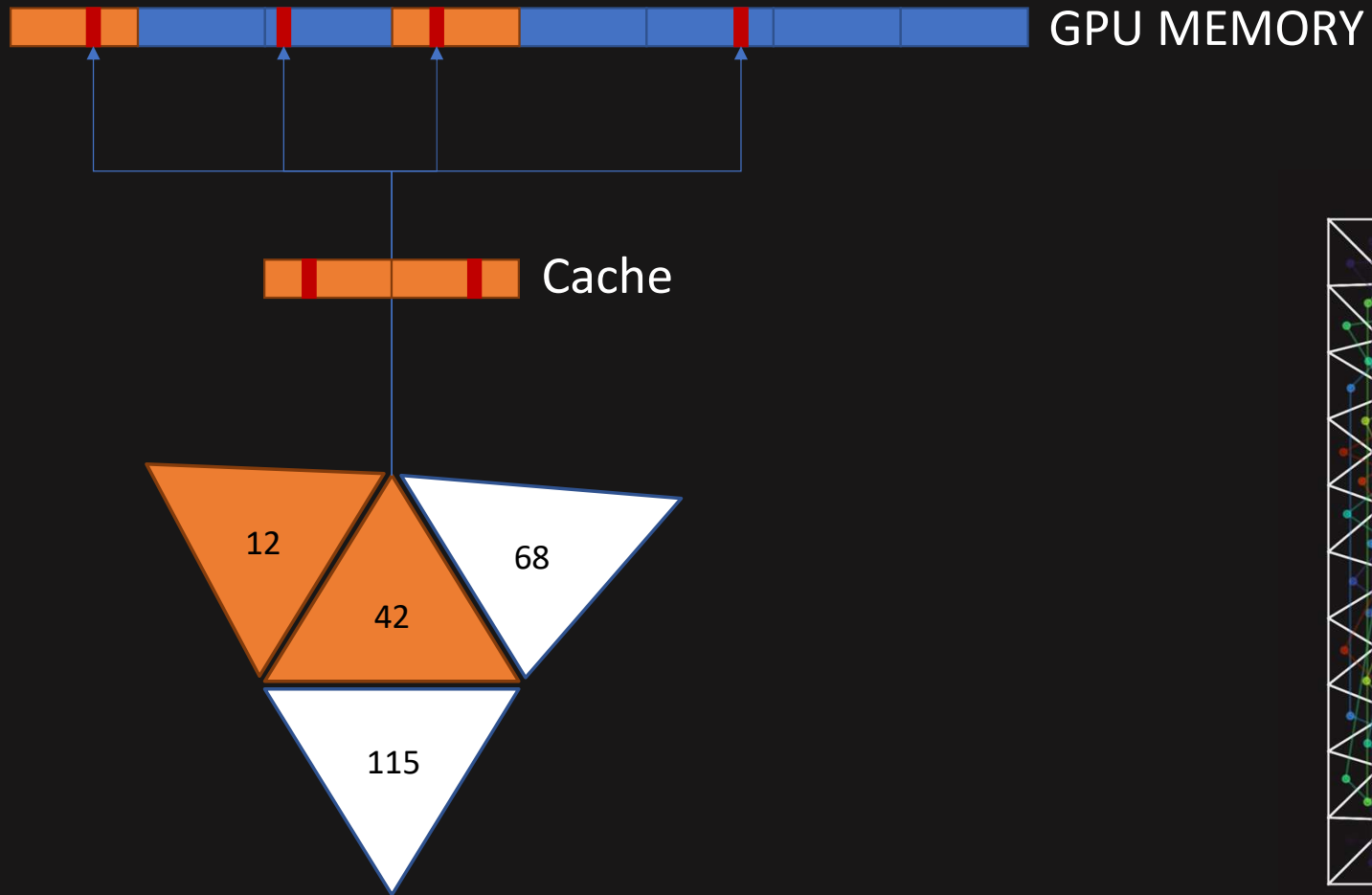


## 2 – Importance of memory locality



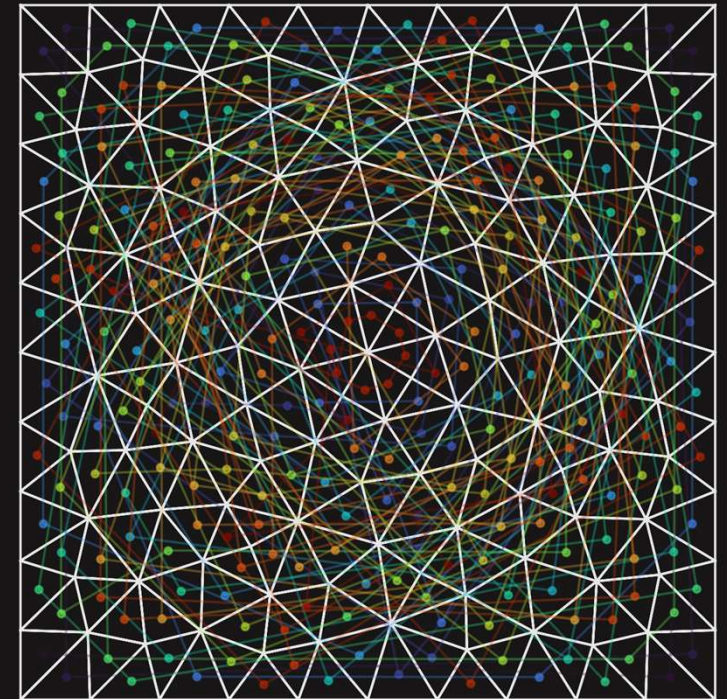
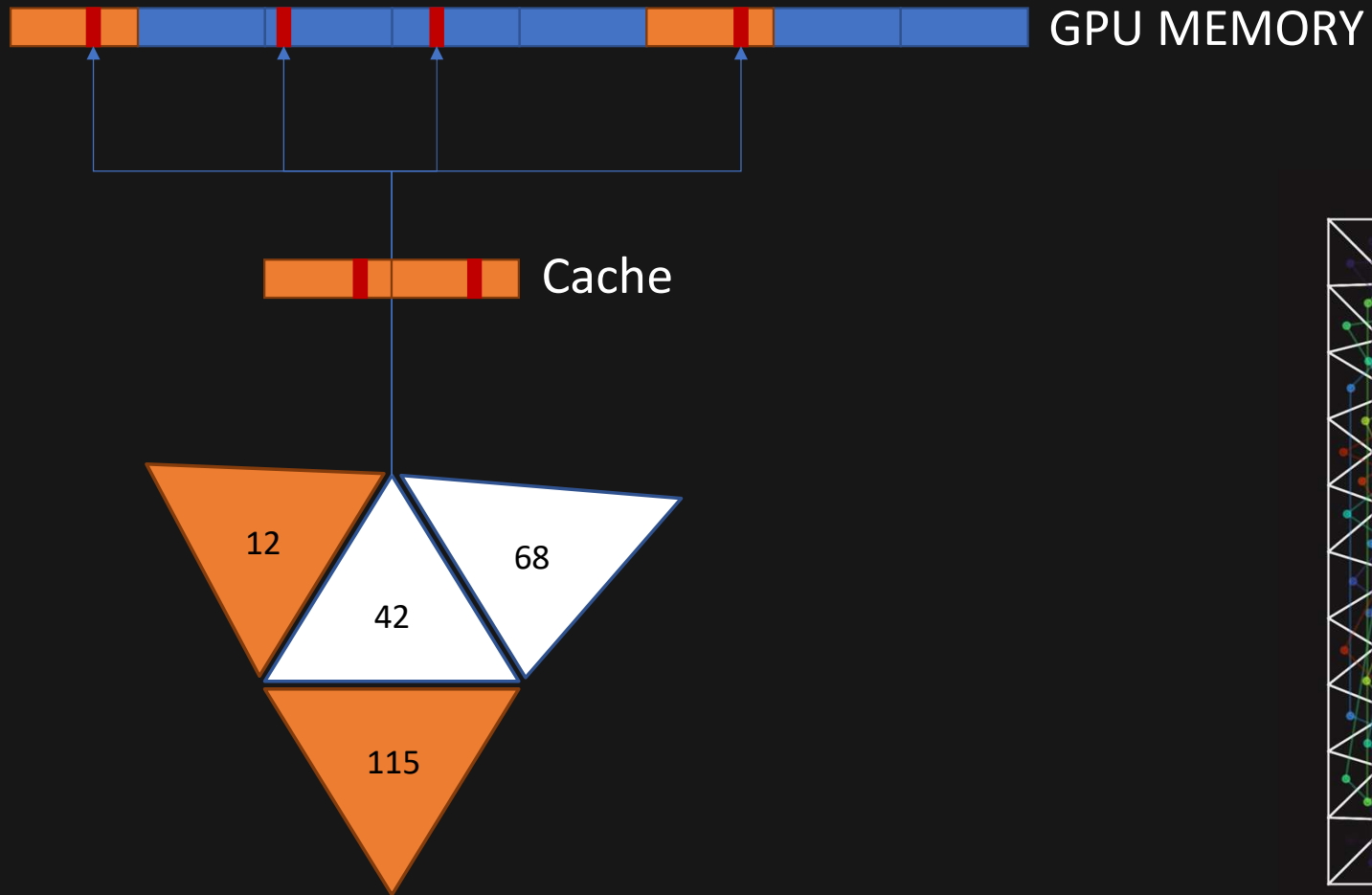


## 2 – Importance of memory locality

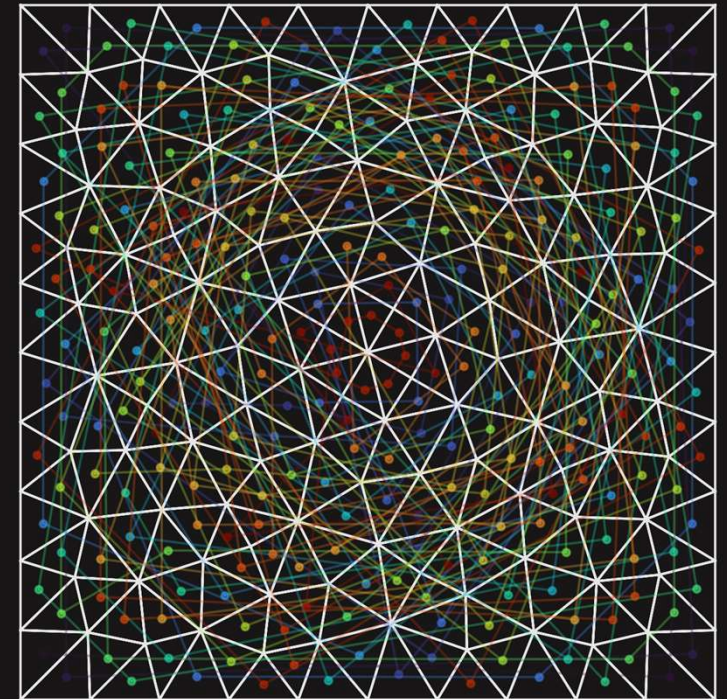
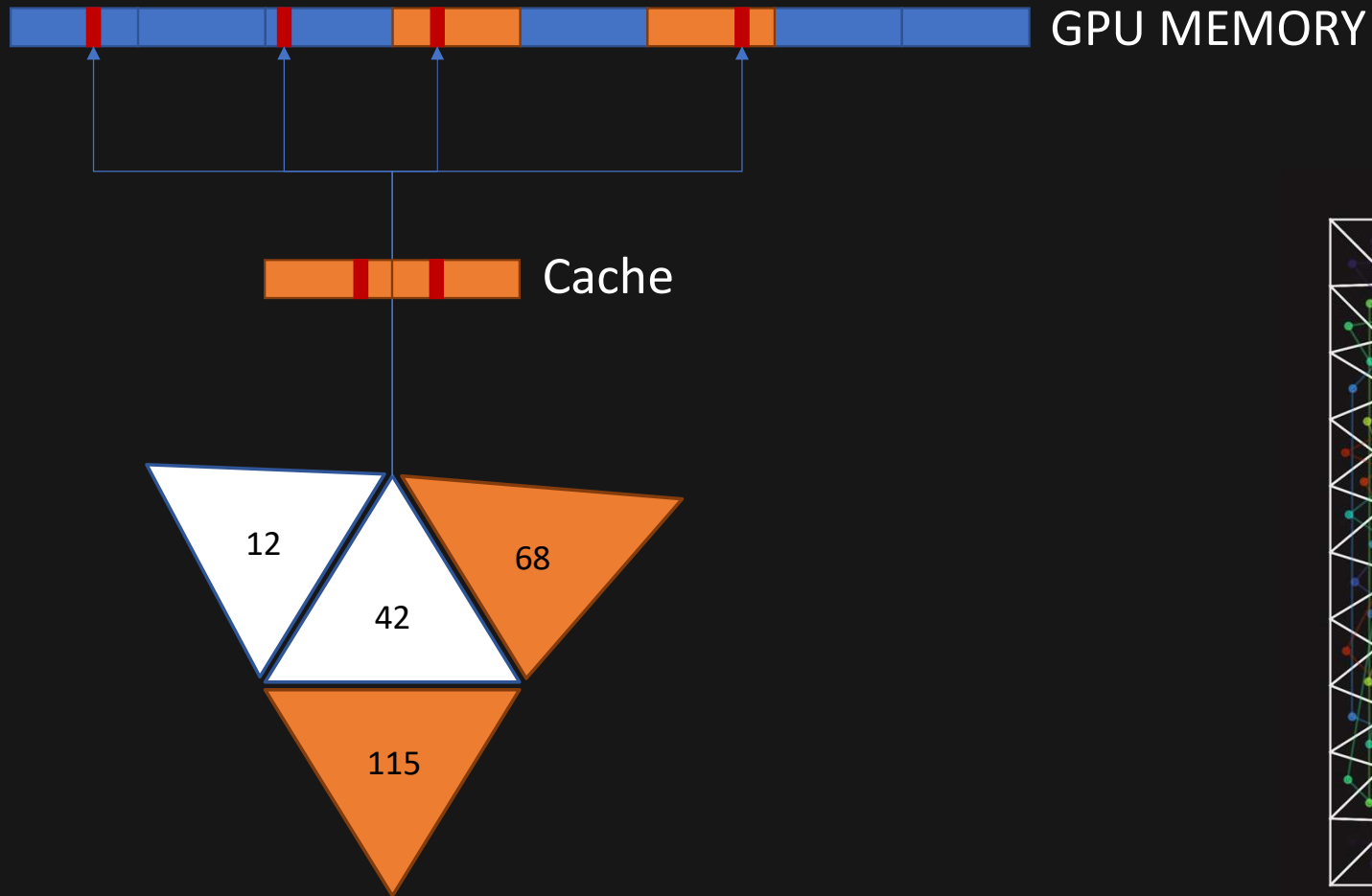




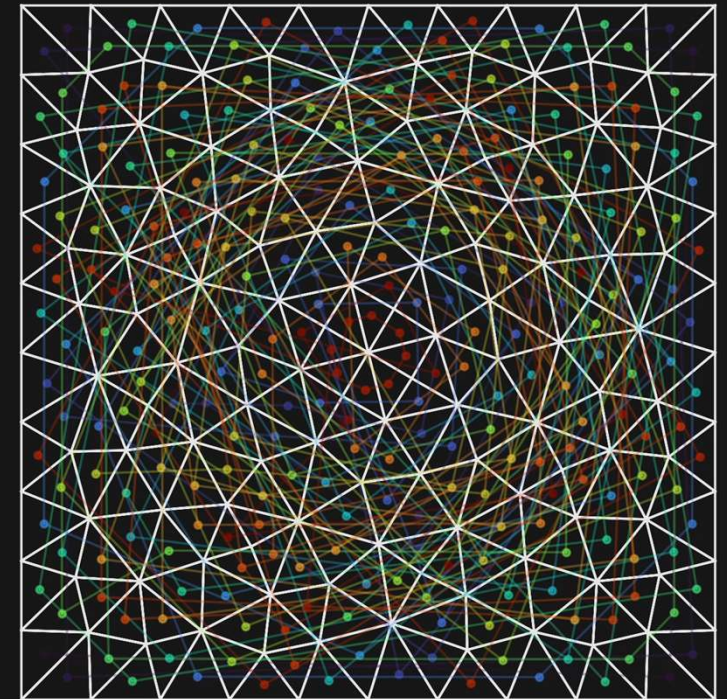
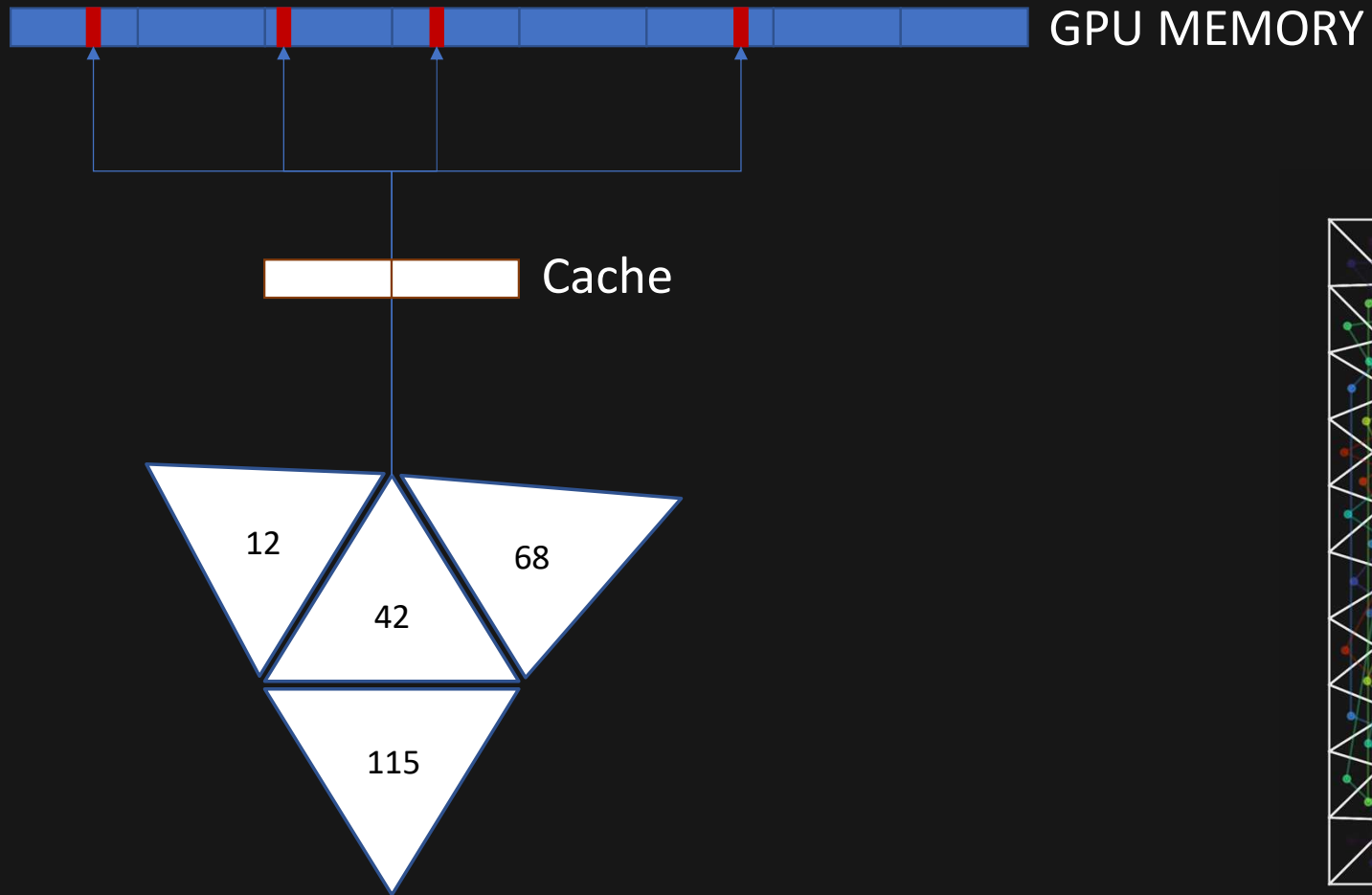
## 2 – Importance of memory locality



## 2 – Importance of memory locality

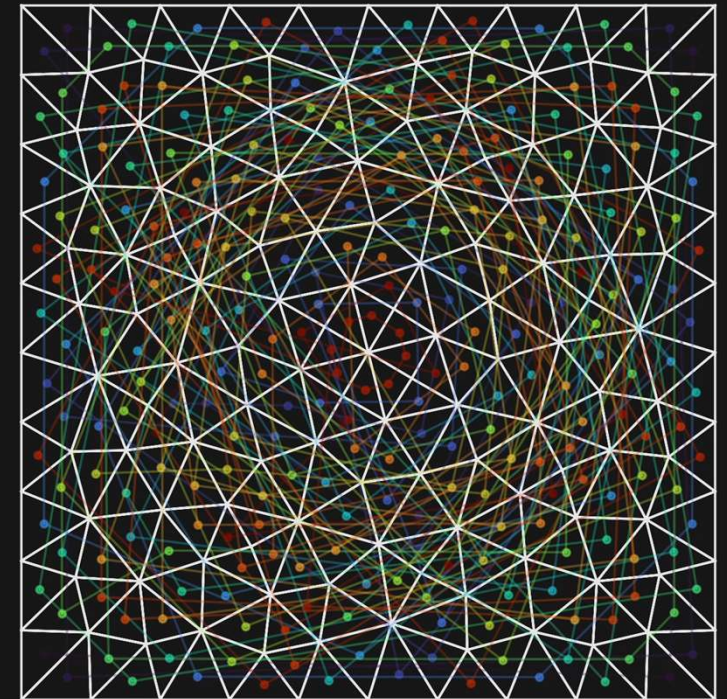
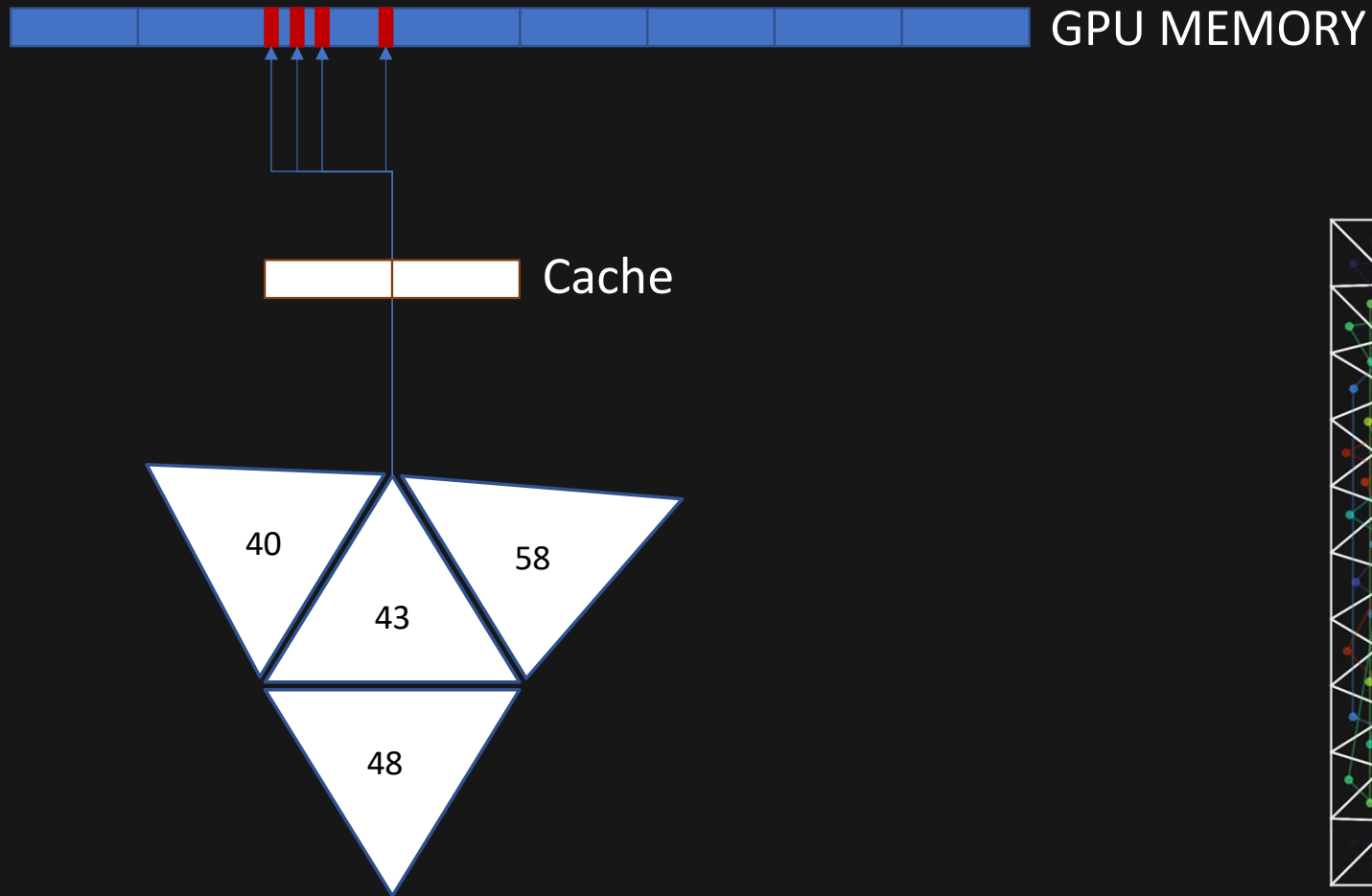


## 2 – Importance of memory locality

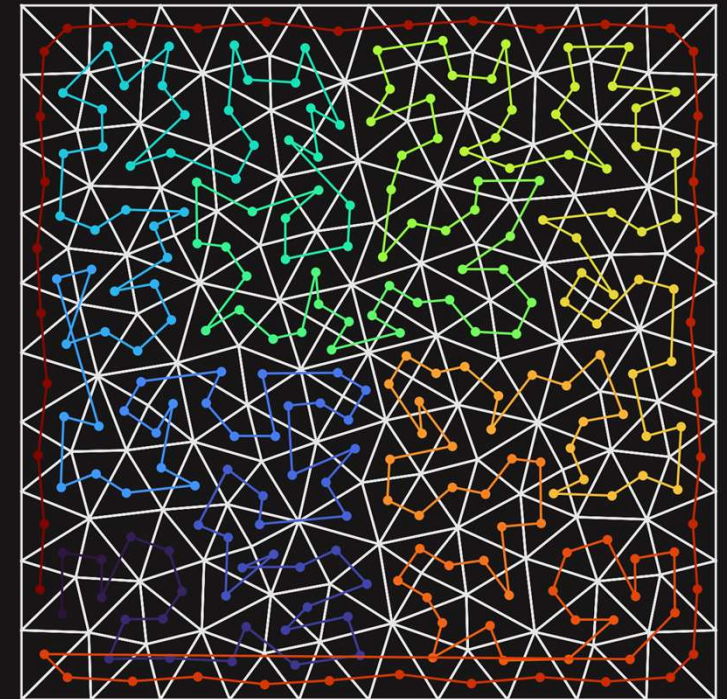
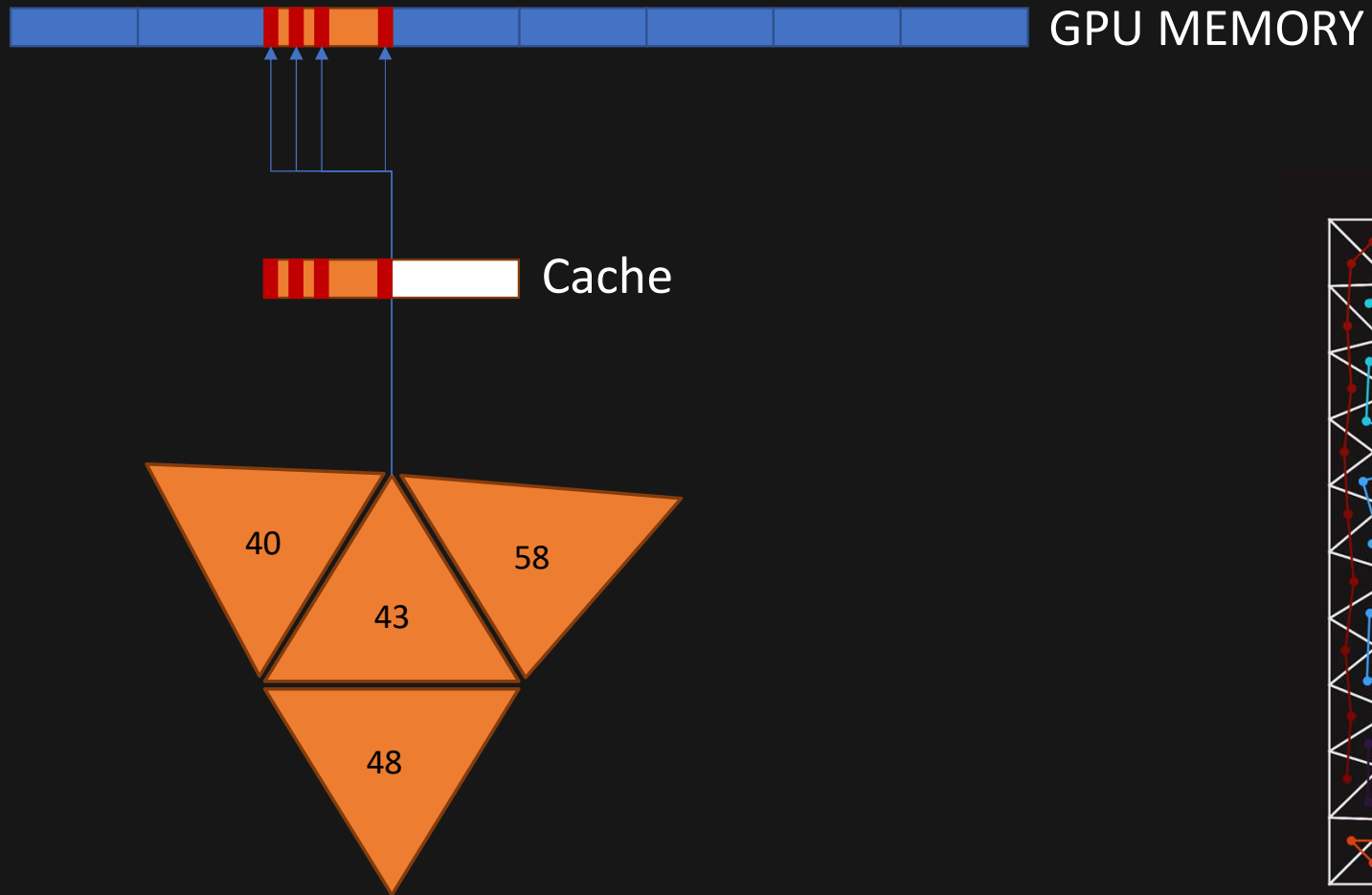




## 2 – Importance of memory locality



## 2 – Importance of memory locality



## 2 – Clues to detect that problem:

- Very high bandwidth utilization
- Too many reads compared to what is expected

**L2 Load Access Pattern** The memory access pattern for loads from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 1.2 sectors out of the possible 4 sectors per cache line. Check the [Source Counters](#) section for uncoalesced loads and try to minimize how many cache lines need to be accessed per memory request.

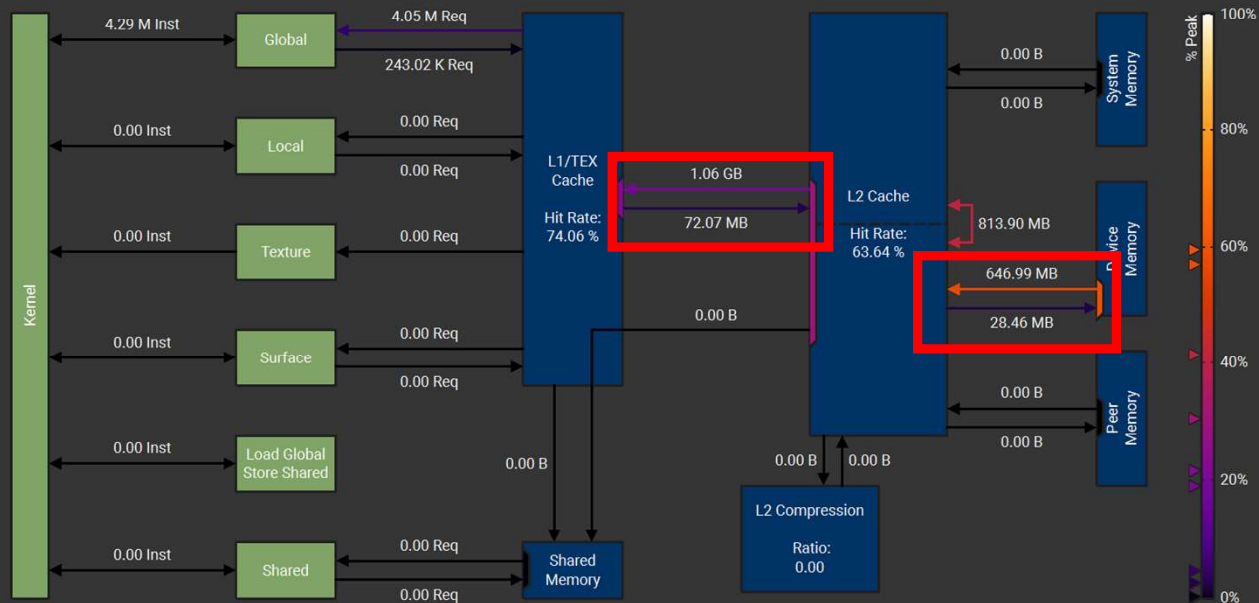
**DRAM Excessive Read Sectors** The memory access pattern for loads from device memory causes 21,733,744 sectors to be read from DRAM, which is 1.8x of the 12,160,697 sectors causing a miss in the L2 cache. The DRAM fetch granularity for read misses in L2 is 64 bytes, i.e. the lower or upper half of an L2 cache line. Try changing your access pattern to make use of both sectors returned by a DRAM read request for optimal usage of the DRAM throughput. For strided memory reads, avoid strides of 64 bytes or larger to avoid moving unused sectors from DRAM to L2.

**Shared Memory Conflicts** Detection of shared memory bank conflicts.

Apply

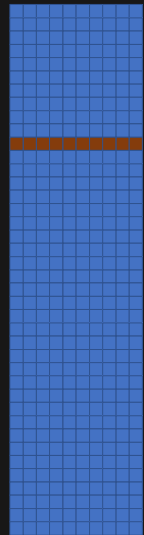
Memory Chart

Show As: Transfer Size

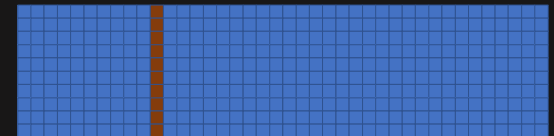


### 3 – So... let's optimize for locality, right ?

```
__global__ void mysum(const float* a, const float* b, float* c, int nf, int n){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n){
        for(int field = 0; field < nf; field++){
            // maximal locality
            c[tid * nf + field] = a[tid * nf + field] + b[tid * nf + field];
        }
    }
}
```



```
__global__ void mysum(const float* a, const float* b, float* c, int nf, int n){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n){
        for(int field = 0; field < nf; field++){
            // very large stride
            c[field * n + tid] = a[field * n + tid] + b[field * n + tid];
        }
    }
}
```



### 3 – No, we need to consider coalescence

```
__global__ void mysum(const float* a, const float* b, float* c, int nf, int n){  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    if (tid < n){  
        for(int field = 0; field < nf; field++){  
            // maximal locality  
            c[tid * nf + field] = a[tid * nf + field] + b[tid * nf + field];  
        }  
    }  
}
```

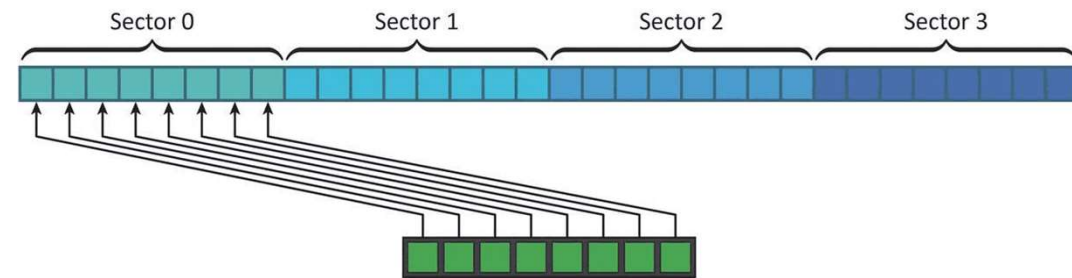
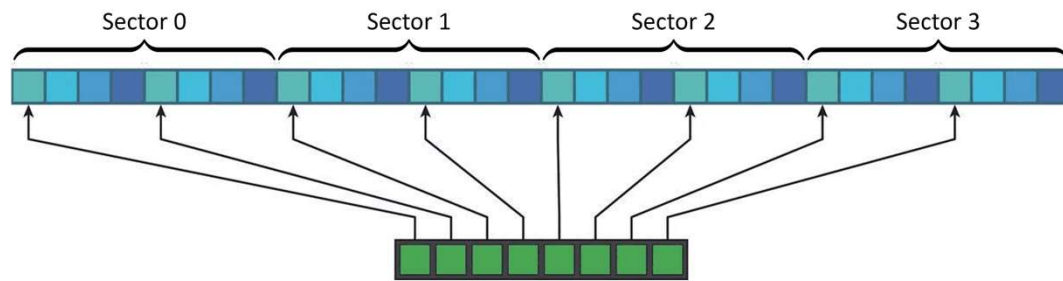
634  $\mu$ s

```
__global__ void mysum(const float* a, const float* b, float* c, int nf, int n){  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    if (tid < n){  
        for(int field = 0; field < nf; field++){  
            // very large stride  
            c[field * n + tid] = a[field * n + tid] + b[field * n + tid];  
        }  
    }  
}
```

403  $\mu$ s



### 3 – No, we need to consider coalescence

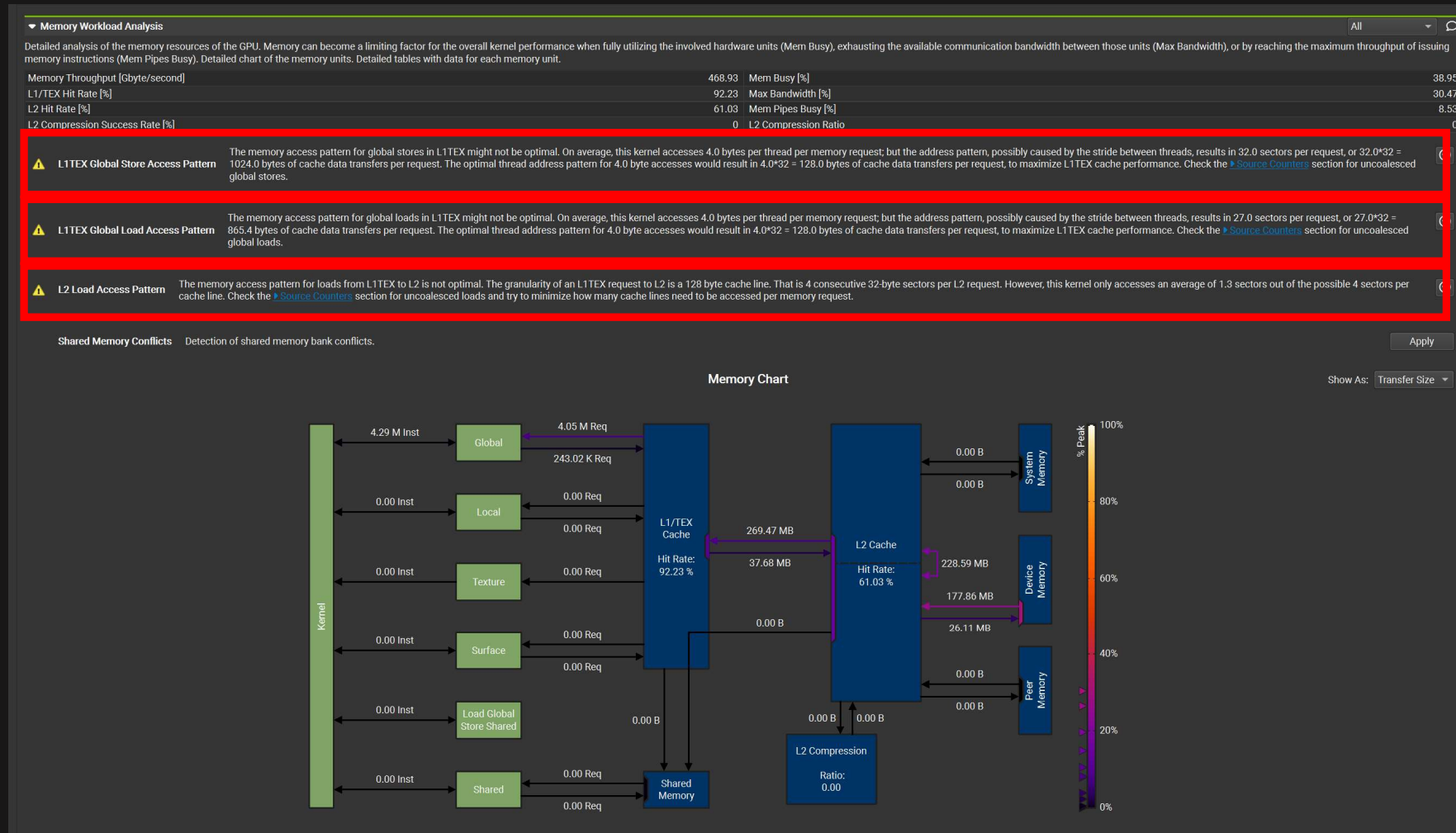


```
__global__ void mysum(const float* a, const float* b, float* c, int nf, int n){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n){
        for(int field = 0; field < nf; field++){
            // maximal locality
            c[tid * nf + field] = a[tid * nf + field] + b[tid * nf + field];
        }
    }
}
```

```
__global__ void mysum(const float* a, const float* b, float* c, int nf, int n){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n){
        for(int field = 0; field < nf; field++){
            // very large stride
            c[field * n + tid] = a[field * n + tid] + b[field * n + tid];
        }
    }
}
```

### 3 – How to evaluate if coalescence is good ?

- ncu will yell at you if it's not



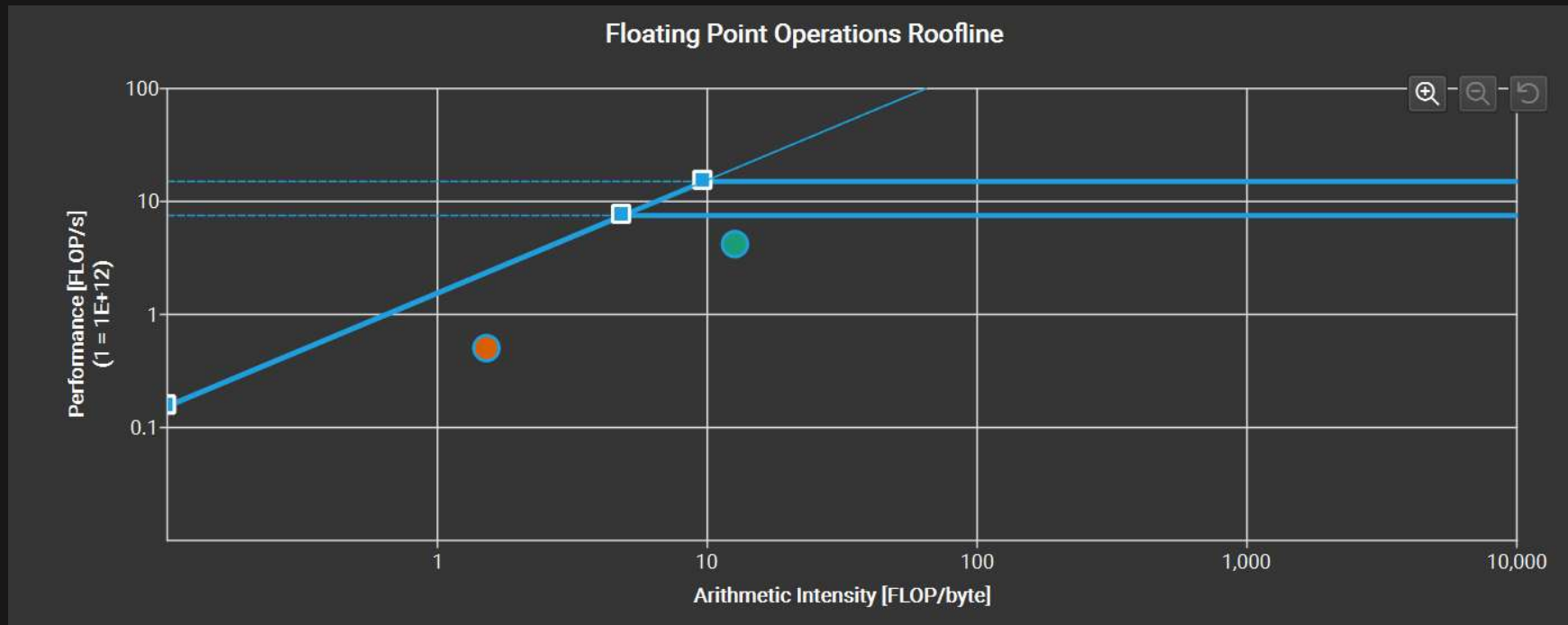
## 4 – Beware of doubles

```
__device__ float inv2x2(const float mat[2][2], float
inv[2][2])
{
    float det = det2x2(mat);
    float ud = 1.0 / det;
    inv[0][0] = mat[1][1] * ud;
    inv[1][0] = -mat[1][0] * ud;
    inv[0][1] = -mat[0][1] * ud;
    inv[1][1] = mat[0][0] * ud;
    return det;
}
```

## 4 – Beware of doubles

```
__device__ float inv2x2(const float mat[2][2], float
inv[2][2])
{
    float det = det2x2(mat);
    float ud = 1.0f / det;
    inv[0][0] = mat[1][1] * ud;
    inv[1][0] = -mat[1][0] * ud;
    inv[0][1] = -mat[0][1] * ud;
    inv[1][1] = mat[0][0] * ud;
    return det;
}
```

## 4 – Beware of doubles : how to check?



## 4 – Beware of doubles : how to check?

- Look for DMUL, DFMA
- Code must be compiled with `--generate-line-info`

The screenshot displays the NVIDIA Nsight Systems interface. The top bar shows the profile name 'profile.ncu-rep'. Below it, the 'Page' is set to 'Source' and the 'Result' is '0 - 520 - dudu'. The 'View' is set to 'Source and SASS'. The 'Source' is 'main.h' and the 'Navigation' is 'Instructions Executed'. The 'DFMA' filter is selected in the top right. The main window is divided into two panes. The left pane shows the source code with a red box highlighting the following lines:

```
...  
// advection  
scalar un = (unl+unr)/2.0;  
flux_t += un*(un > 0 ? Hurl : Hutr);  
flux_n += 0.5*(Hurl*unl + Hutr*unr);  
...  
// gravity  
flux_H += Hun + c*(etal-etar)/2.0;  
flux_n -= g*(bath_l + (etal+etar)/2.0)*(etal-etar)/  
...  
scalar flux_u = flux_n*nx+flux_t*tx;  
scalar flux_v = flux_n*ny+flux_t*ty;  
...  
flux[0] = flux_H;  
flux[1] = flux_u;  
flux[2] = flux_v;  
...  
////////// END OF THE EXERCISE, NO CHANGES NEEDED BELOW
```

The right pane shows the assembly instructions with a red box highlighting the following lines:

```
406 00007fe9 62fbb550 MOV R4, 0x1870  
407 00007fe9 62fbb560 CALL.REL.NOINC 0x7fe962fbb560  
408 00007fe9 62fbb570 MOV R17, R43  
409 00007fe9 62fbb580 BSYNC B0  
410 00007fe9 62fbb590 FMUL R8, R8, R17  
411 00007fe9 62fbb5a0 F2F.F64.F32 R12, R22  
412 00007fe9 62fbb5b0 FADD R11, R18, R3  
413 00007fe9 62fbb5c0 FFMA R8, R7, R10, R8  
414 00007fe9 62fbb5d0 FADD R18, R18, -R3  
415 00007fe9 62fbb5e0 FADD R10, R17, R10  
416 00007fe9 62fbb5f0 F2F.F64.F32 R4, R8  
417 00007fe9 62fbb600 FMUL R9, R9, R18  
418 00007fe9 62fbb610 FMUL R10, R10, 0.5  
419 00007fe9 62fbb620 F2F.F64.F32 R20, R23  
420 00007fe9 62fbb630 FSETP.GT.AND P0, PT, R10, R2, PT  
421 00007fe9 62fbb640 F2F.F64.F32 R14, R14, 0.5  
422 00007fe9 62fbb650 F2F.F64.F32 R14, R14  
423 00007fe9 62fbb660 FMUL R14, R94, R14  
424 00007fe9 62fbb670 F2F.F64.F32 R6, R6  
425 00007fe9 62fbb680 DMUL R12, R14, R12  
426 00007fe9 62fbb690 F2F.F64.F32 R32, R4  
427 00007fe9 62fbb6a0 F2F.F64.F32 R8, R9  
428 00007fe9 62fbb6b0 DFMA R12, R12, -0.5, R32  
429 00007fe9 62fbb6c0 DFMA R6, R8, 0.5, R6  
430 00007fe9 62fbb6d0 F2F.F32.F64 R12, R12
```

The bottom of the interface shows the 'Inline Functions' and 'Source Markers' tabs, with a note: 'Select a source line in an active inline function to show additional information.'

## 5 – Why did a simple `printf` make my code 1.2x slower ?

```
__device__ float inv2x2(const float mat[2][2], float
inv[2][2])
{
    float det = det2x2(mat);
    if(det){
        float ud = 1.0 / det;
        inv[0][0] = mat[1][1] * ud;
        inv[1][0] = -mat[1][0] * ud;
        inv[0][1] = -mat[0][1] * ud;
        inv[1][1] = mat[0][0] * ud;
    }
    else{
        printf("Singular matrix 2x2");
        for(int i = 0; i < 2; i++)
            for(int j = 0; j < 2; j++)
                inv[i][j] = 0.0;
    }
    return det;
}
```

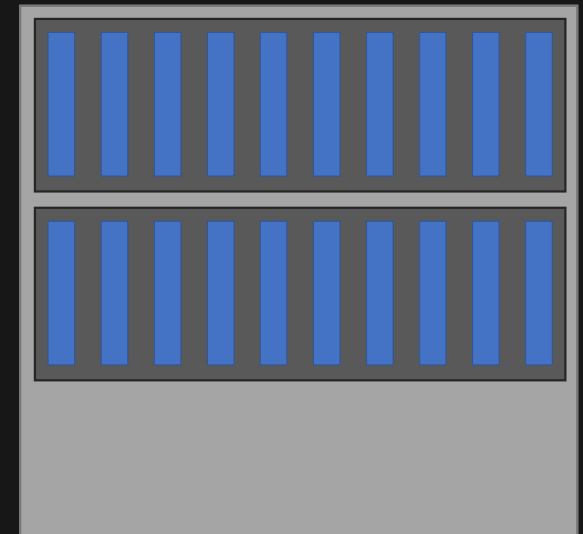
## 5 – Why did a simple `printf` make my code 1.2x slower ?

```
__device__ float inv2x2(const float mat[2][2], float
inv[2][2])
{
    float det = det2x2(mat);
    float ud = 1.0 / det;
    inv[0][0] = mat[1][1] * ud;
    inv[1][0] = -mat[1][0] * ud;
    inv[0][1] = -mat[0][1] * ud;
    inv[1][1] = mat[0][0] * ud;
    return det;
}
```



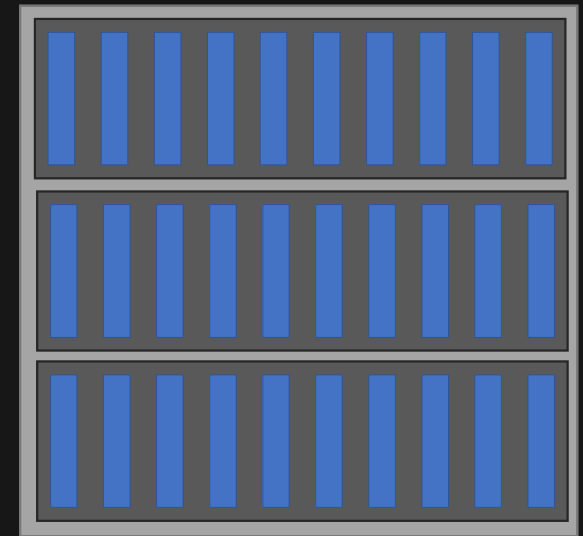
## 5 – Watch the occupancy!

```
__device__ float inv2x2(const float mat[2][2], float
inv[2][2])
{
    float det = det2x2(mat);
    if(det){
        float ud = 1.0 / det;
        inv[0][0] = mat[1][1] * ud;
        inv[1][0] = -mat[1][0] * ud;
        inv[0][1] = -mat[0][1] * ud;
        inv[1][1] = mat[0][0] * ud;
    }
    else{
        printf("Singular matrix 2x2");
        for(int i = 0; i < 2; i++)
            for(int j = 0; j < 2; j++)
                inv[i][j] = 0.0;
    }
    return det;
}
```



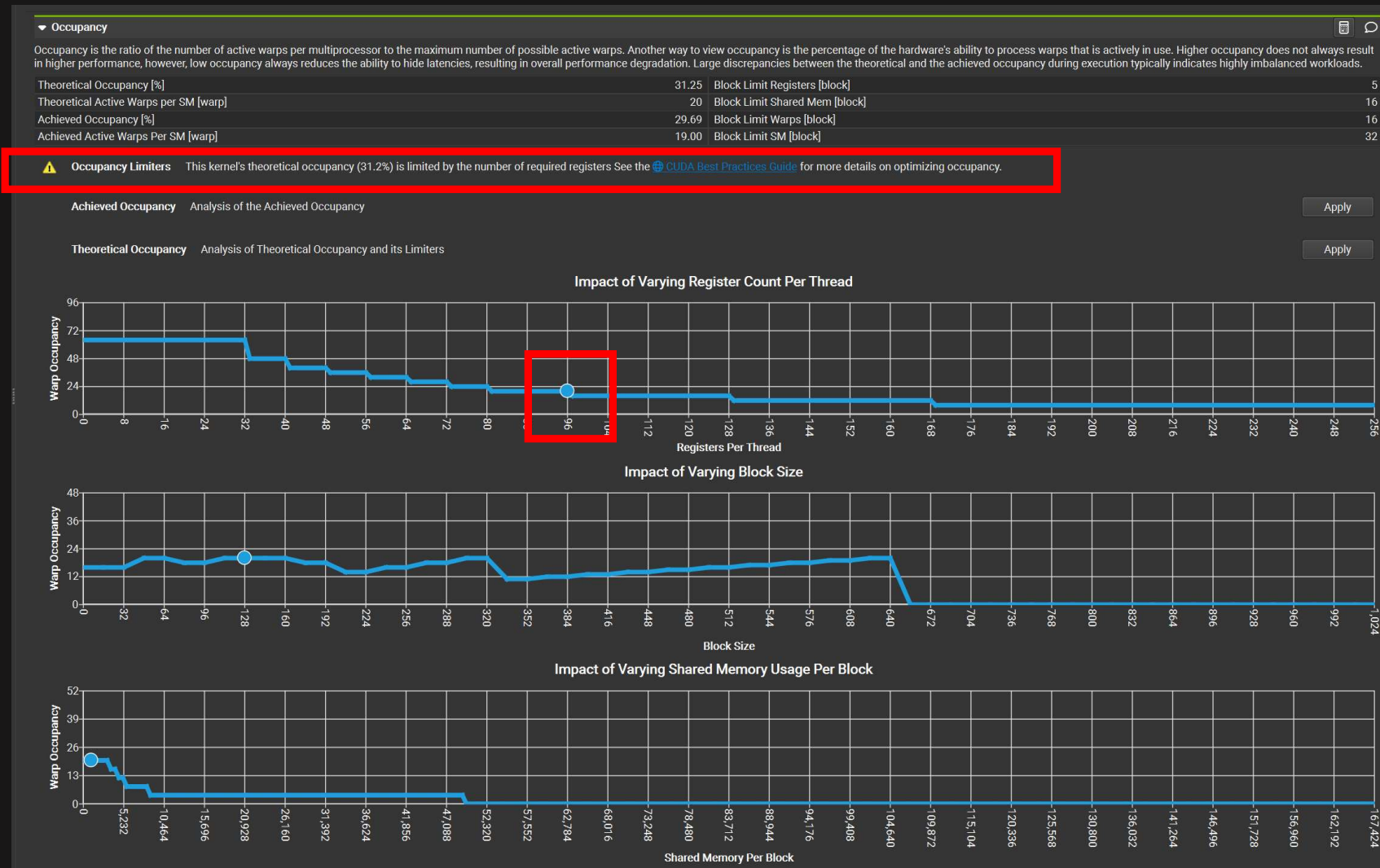
## 5 – Watch the occupancy!

```
__device__ float inv2x2(const float mat[2][2], float  
inv[2][2])  
{  
    float det = det2x2(mat);  
    float ud = 1.0 / det;  
    inv[0][0] = mat[1][1] * ud;  
    inv[1][0] = -mat[1][0] * ud;  
    inv[0][1] = -mat[0][1] * ud;  
    inv[1][1] = mat[0][0] * ud;  
    return det;  
}
```



## 5 – Is the occupancy a limiting factor?

- ncu will tell you
- Often the case that occupancy is limited by registers, but critical when « on a step »
- Can sometimes be mitigated by template kernel specialization



# Exercise time!

1. Use floats
2. Use a reordered mesh
3. Transpose the memory access for more coalescence (at the price of locality though!)
4. Catch the remaining double literals
5. Remove the never-accessed debug print code

File : main.h

## 6 – Kernel fusion

- Kernels comparable in duration
  - Kernels share variables
- Makes sense to merge the kernels to avoid a read and write from global memory



A horizontal timeline diagram consisting of four blue rectangular segments. The first segment is labeled 'axpy', the second segment is labeled 'dudt', and the third segment is labeled 'axpy'. The fourth segment is empty. The segments are separated by thin vertical lines.

axpy

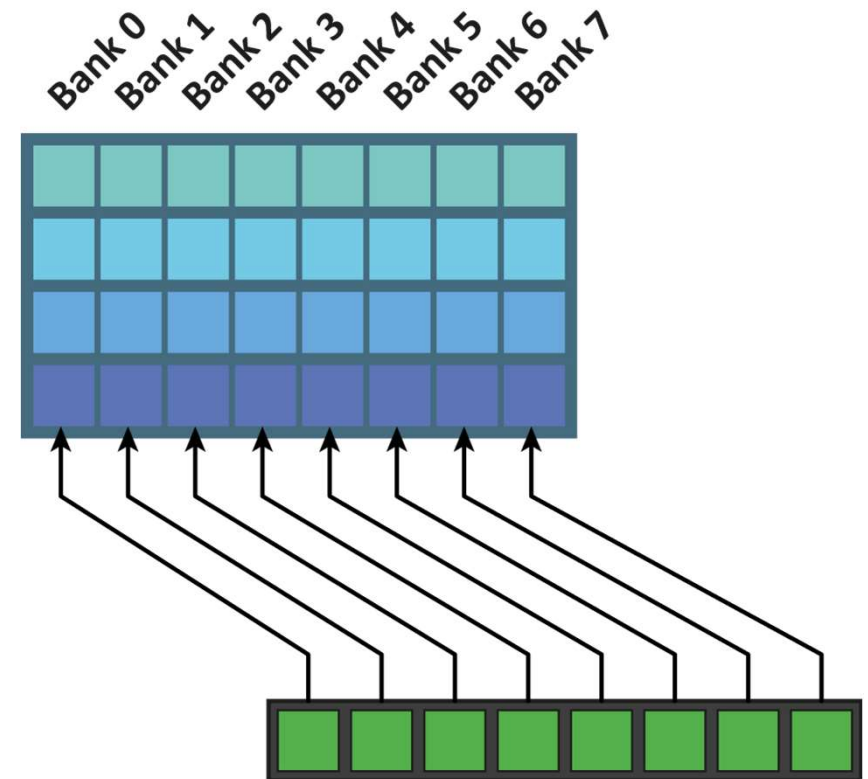
dudt

axpy

Timeline

## 7 – Using shared memory

- To share data among threads
  - **To manually cache some frequently used data**
  - To reduce register pressure/ local memory usage
  - To allow communication/ data exchange within a group
- 
- Shared memory organized in « Banks »
  - Simultaneous accesses to the same bank are serialized
  - Consecutive threads should access consecutive banks



## 8 – Array of struct of arrays

- Array of struct : perfect locality, bad coalescence



- Struct of array : good coalescence, bad locality



- What if we could combine both?



Array-of-struct-of-array layout : good coalescence, fairly good locality

## 9 – Free performance\*?

- `--use_fast_math` : compiler flag that enables **unsafe** and less accurate but sometimes faster math (Can reduce register usage significantly)
- `--extra-device-vectorization`
- `__launch_bounds__()` : Tell the compiler the maximum block size at compile time. Allow optimization that can significantly improve the performance, **or sometimes significantly worsen the performance**.

Usage :

```
__launch_bounds__(BLOCK_SIZE)
__global__ void my_kernel(float a, float* data, int n){...
```

\*Sometimes



**All of these are just ideas:**

**Some things may work and improve the performance,**

**Others may slow down the code.**

**Some can degrade the quality of the solution,**

**Some may not be possible due to resource constraints,  
etc...**

**Conclusion :**

**profile, benchmark and run your code before and you try to optimize it.**

**You never know.**