GPU optimization techniques and tools

A collection of ideas to maybe improve your GPU performance

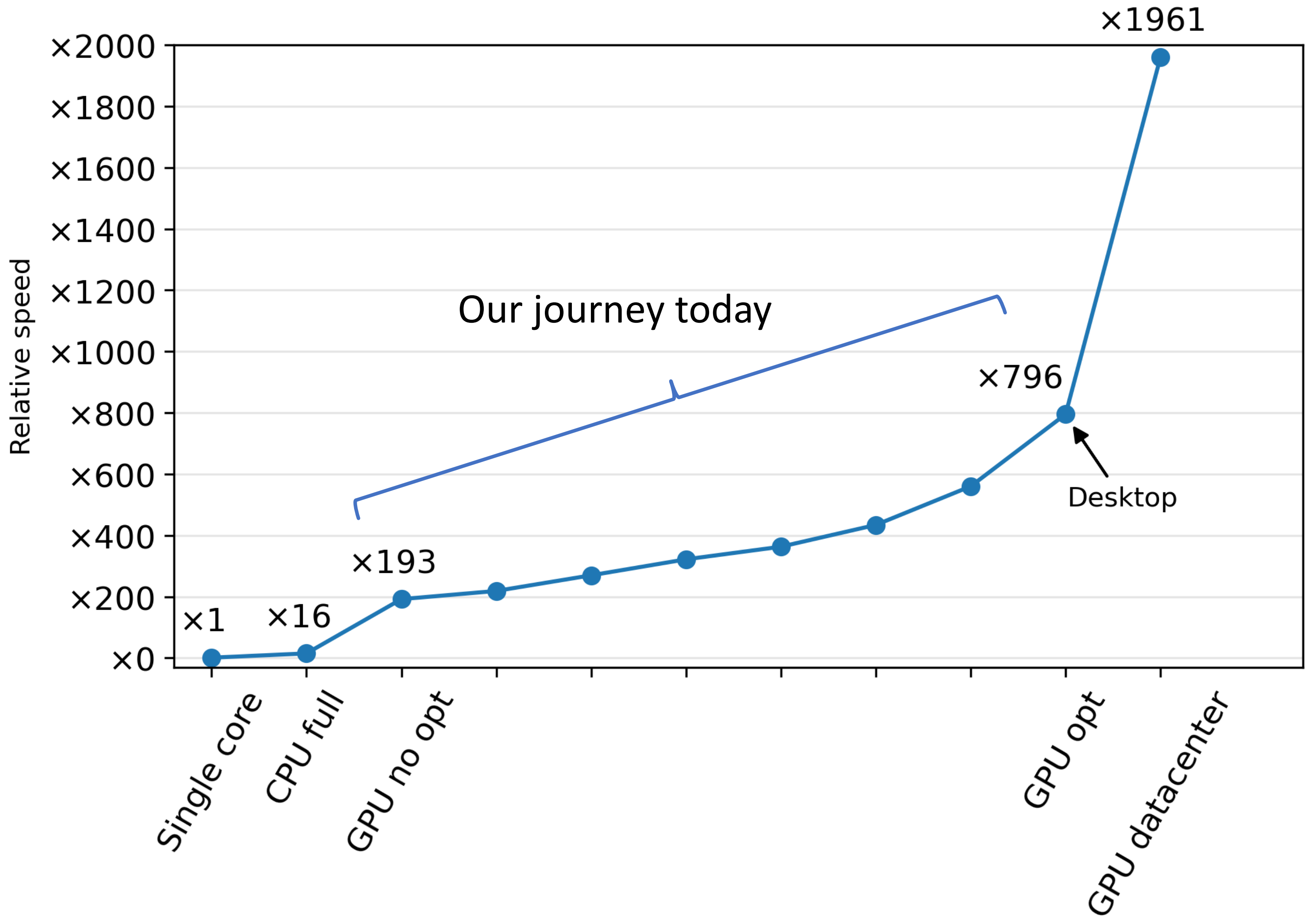
*Presented data acquired over a two years period of hands-on experimentation and accumulated frustration and suffering.

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



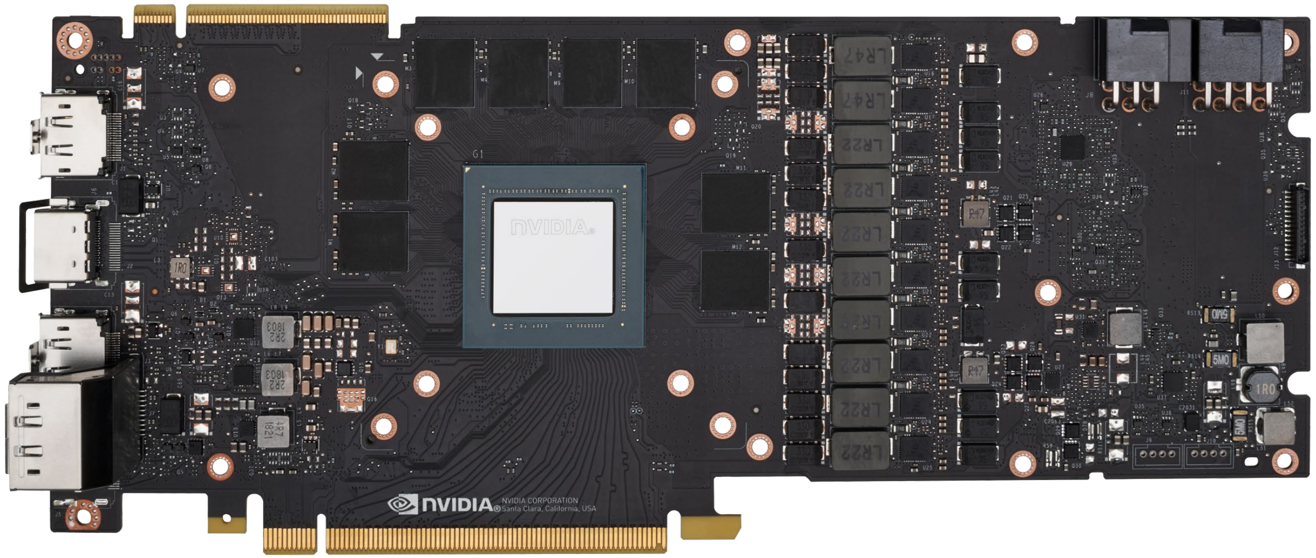
Plan for this session

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
- …

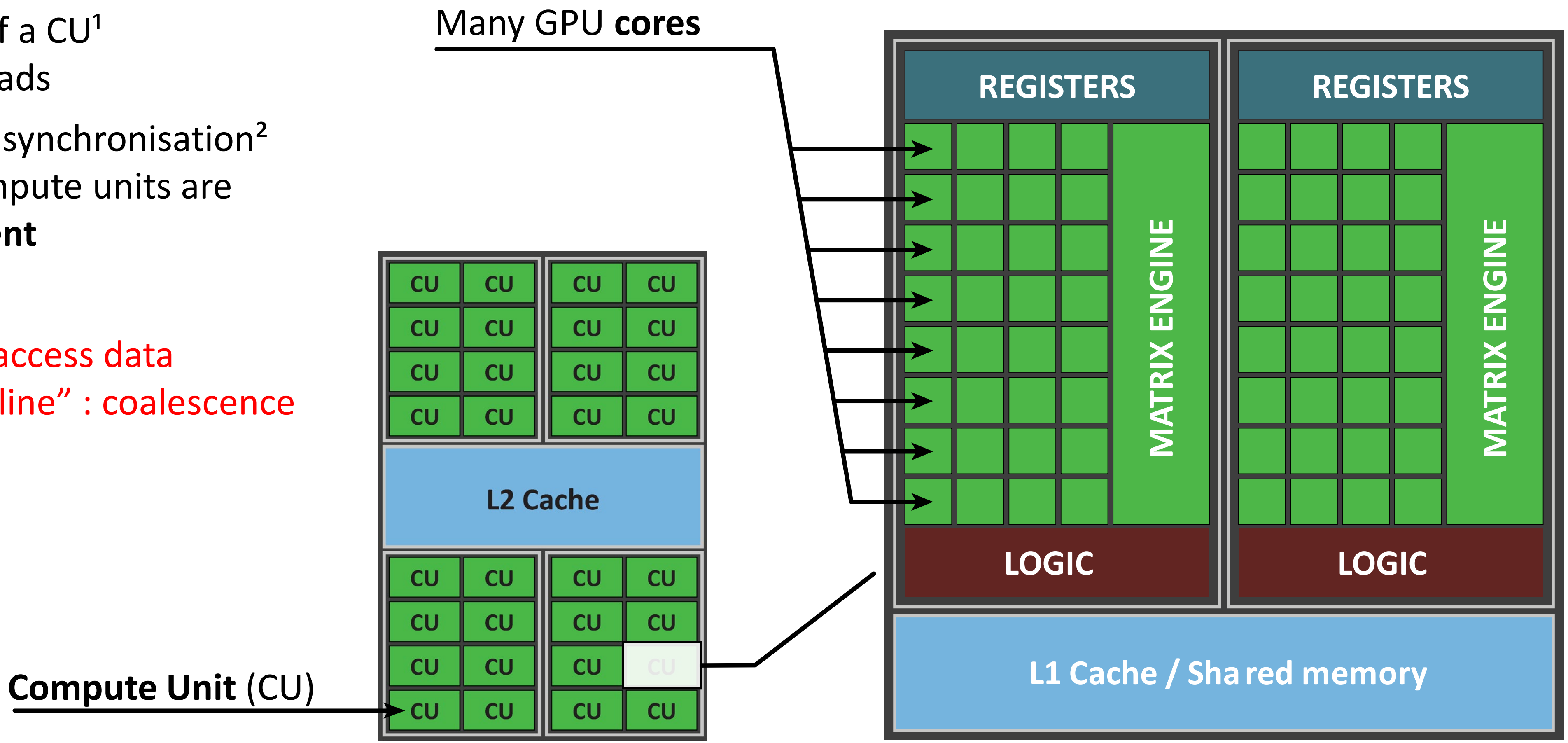


What is a GPU : the execution model

Defining characteristics:

- Cores of a CU[1] are **not independent**
- Computations inside of a CU[1] is the same for all threads
- No data exchange and synchronisation[2] outside of the CU. Compute units are **completely independent**

- Cores in a CU want to access data from the same "cache line" : coalescence



Simplified GPU

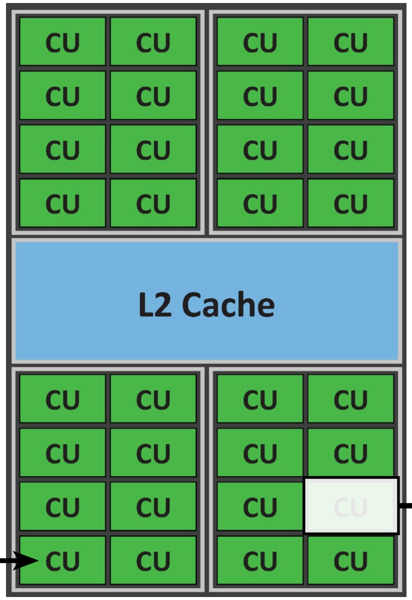
Zoom on a compute unit

Nomenclature

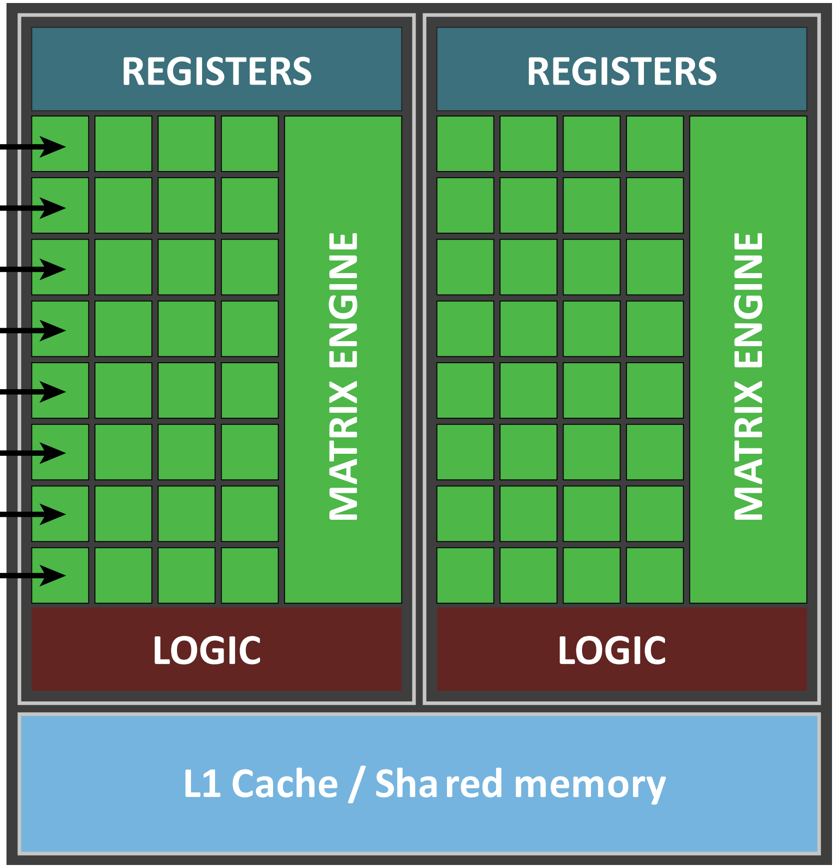
Green = NVIDIA
Blue = INTEL
Red = AMD

(Cuda) cores = Shading Units = Stream processors

Streaming multiprocessor = Core = Compute Unit



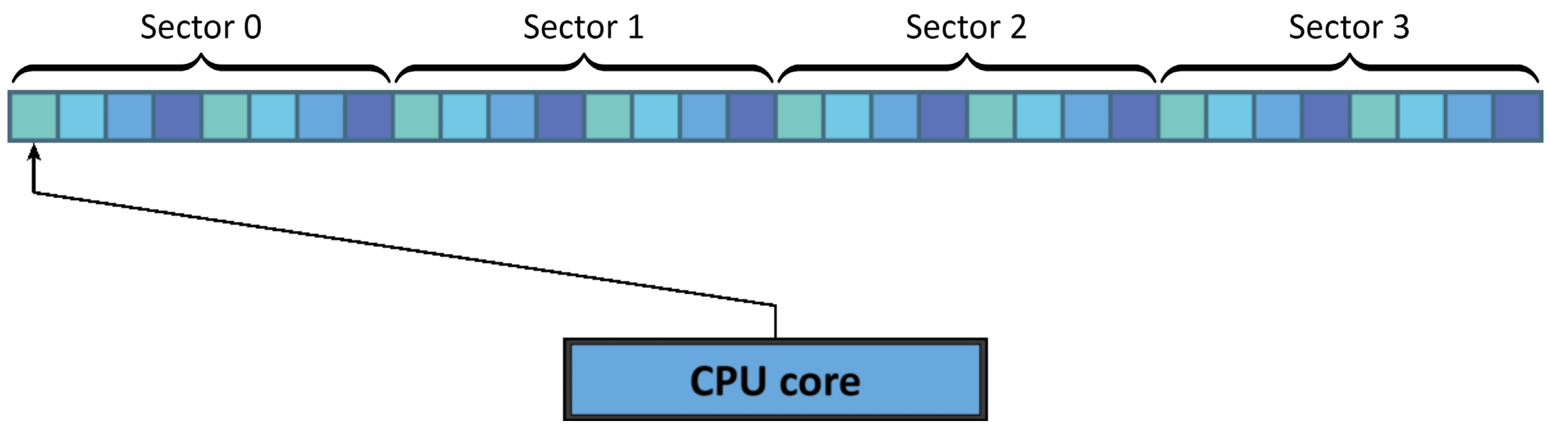
Simplified GPU



Zoom on a compute unit

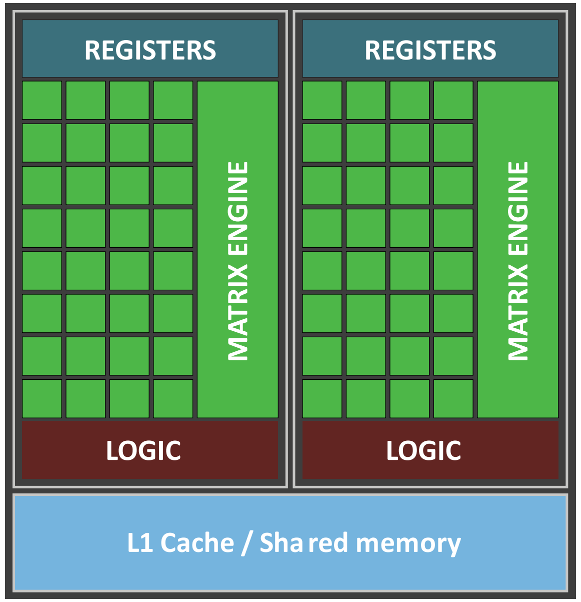
Coalescent memory access

- On the CPU : we want to maximize locality

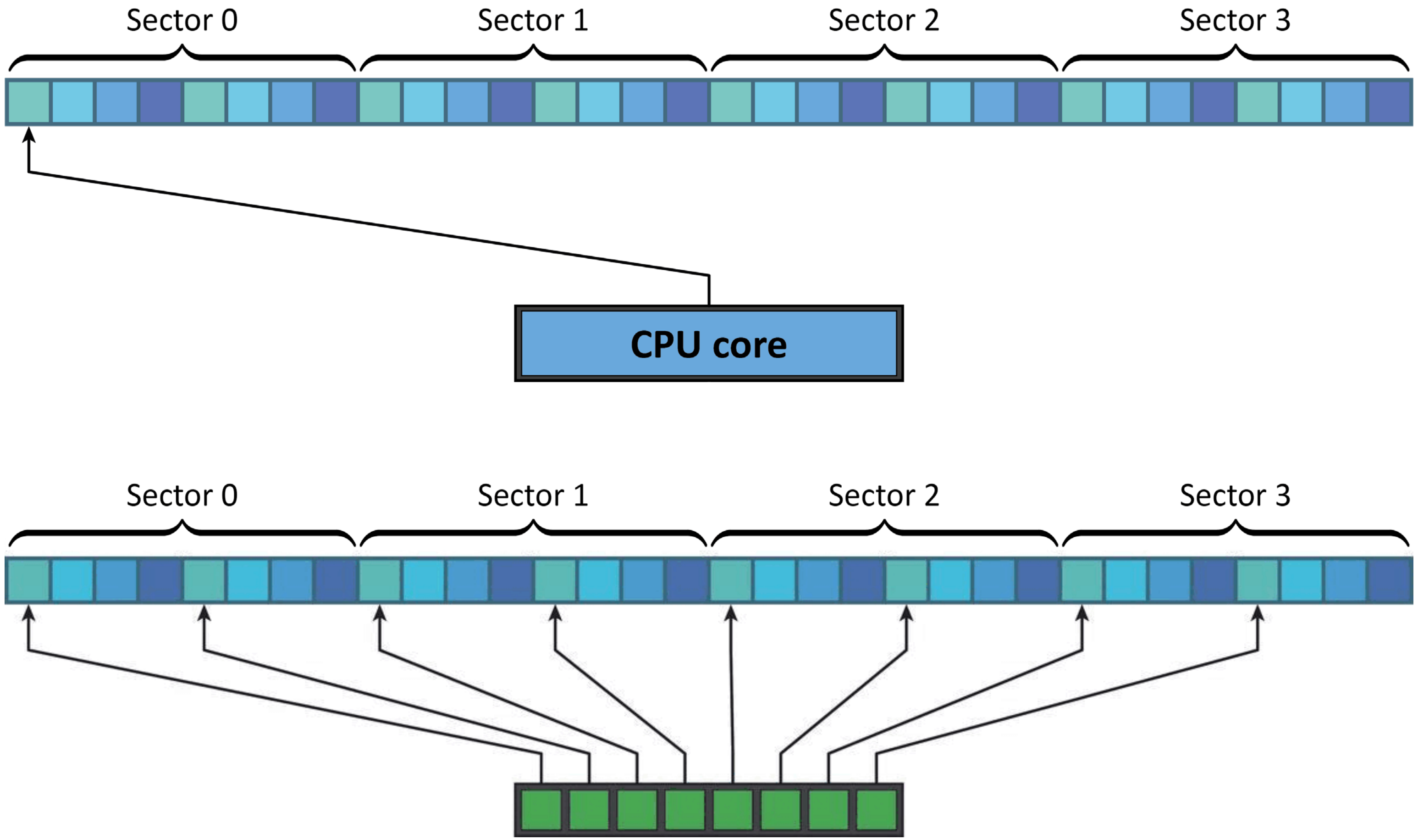


Coalescent memory access

- 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

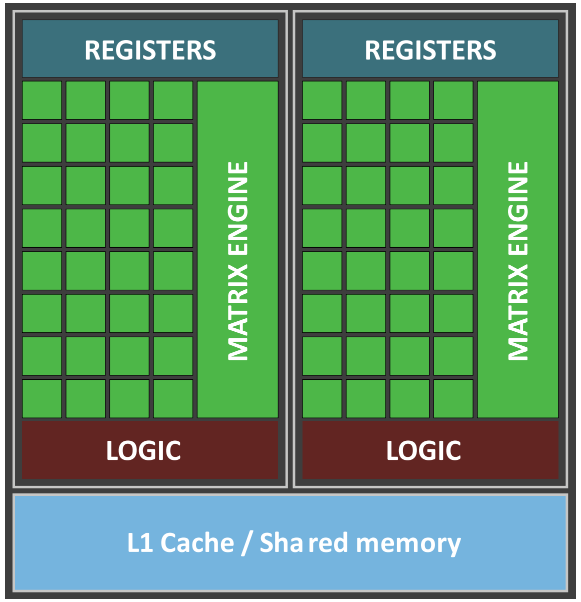


compute unit

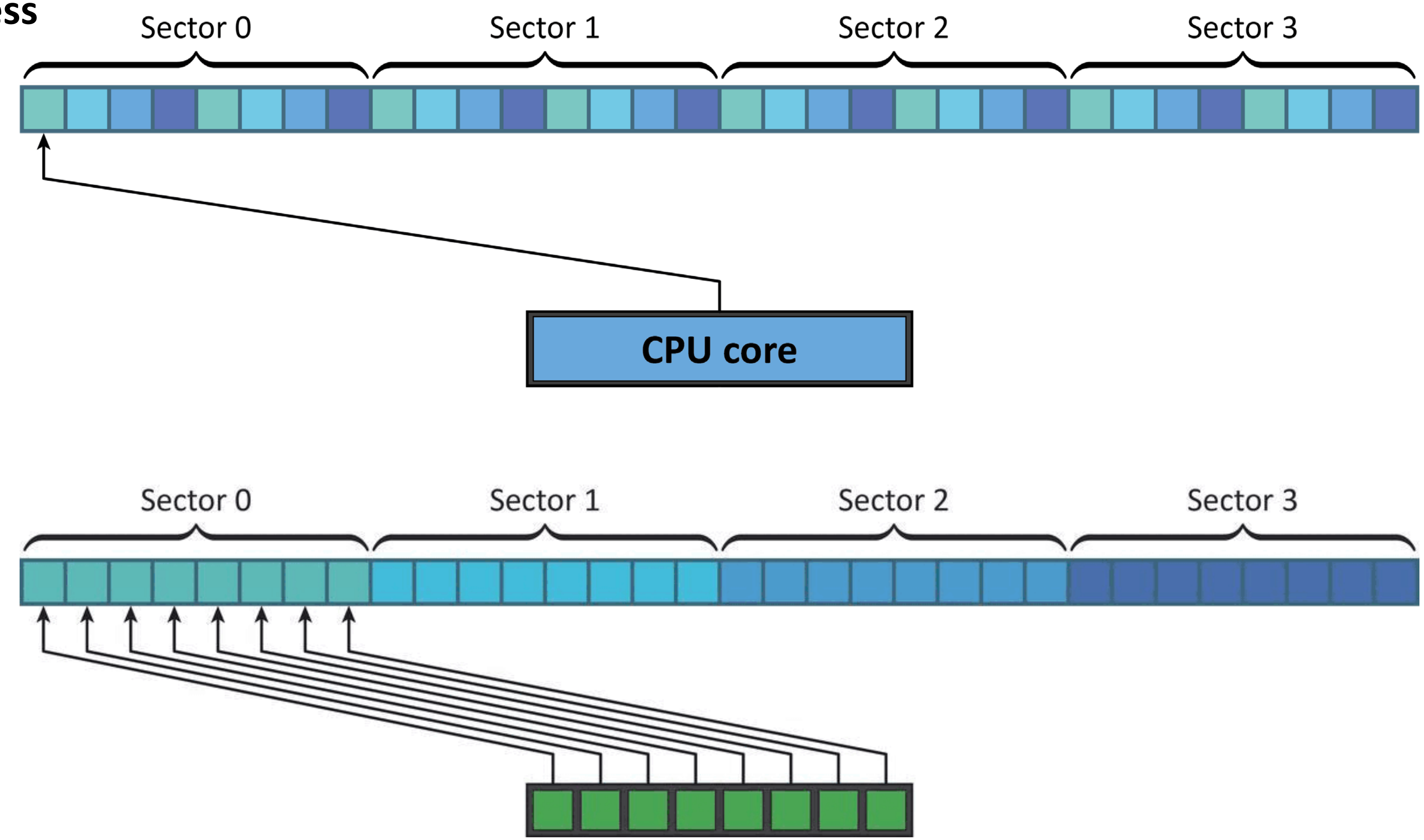


Coalescent memory access

- 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**

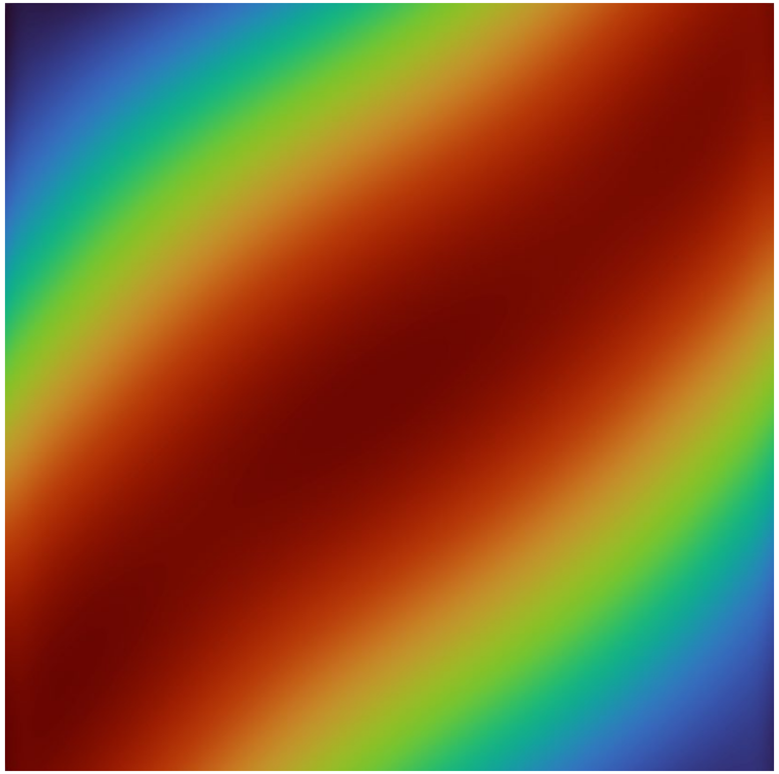


compute unit



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

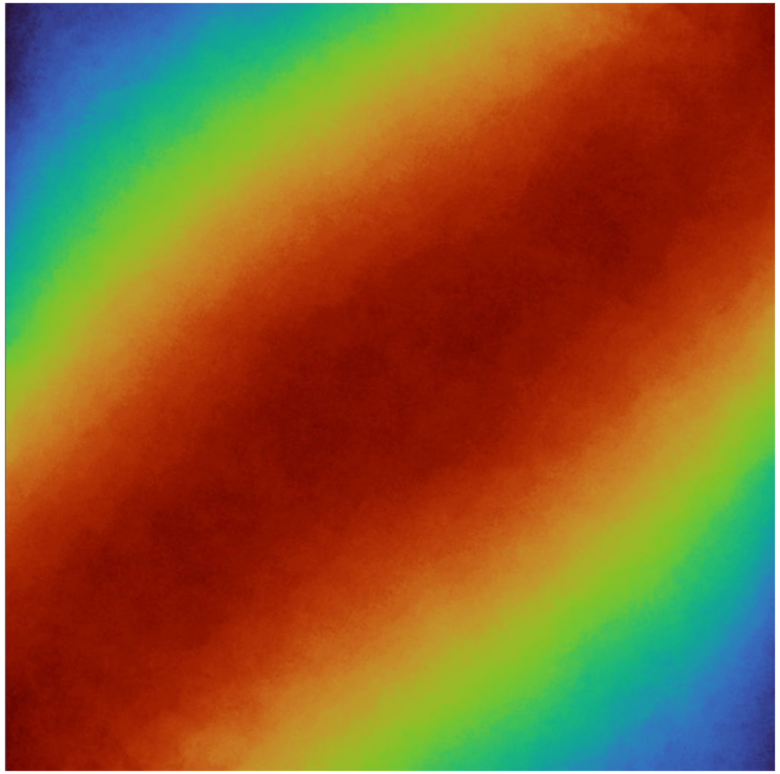
- RTX3080:
  - Float : 29.77 TFLOPS
  - Double : 0.47 TFLOPS
- A100:
  - Float : 19.49 TFLOPS
  - Double : 9.746 TFLOPS



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;                          } 4%
        local_u[...] = u[...];

        // 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
```

Heavy processing

```
        // writing back the result
        fu[...] = local_fu;
    }
}
```

Memory is (almost always) the bottleneck

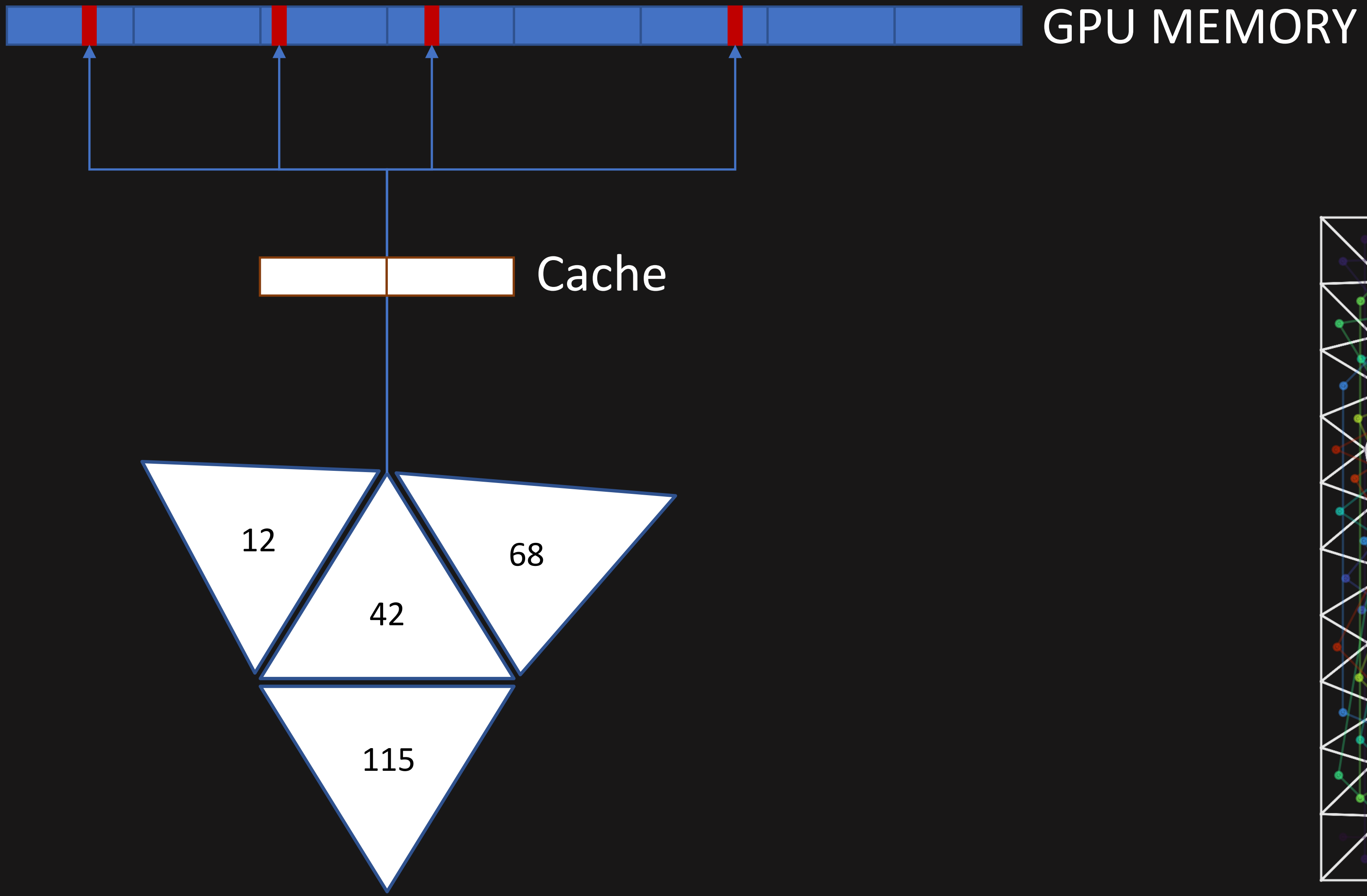
```
__global__ void dudt_kernel(const float* u, float* fu, ...){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n){
        // loading data
        float[4][3] local_u;
        local_u[...] = u[...];

        // some heavy computation
        Heavy processing

        // writing back the result
        fu[...] = local_fu;
    }
}
```

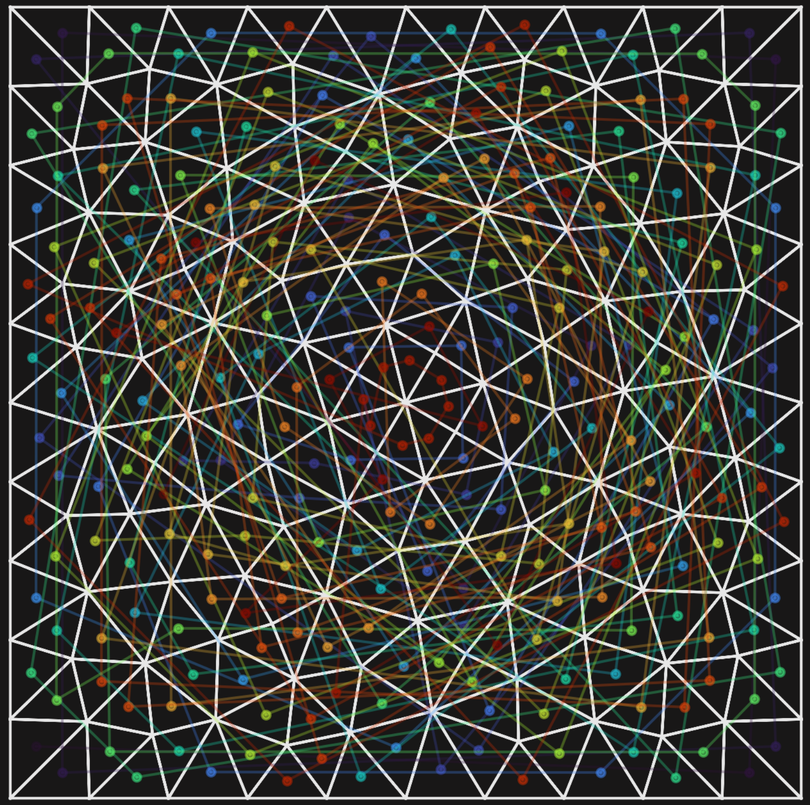
55%

30%

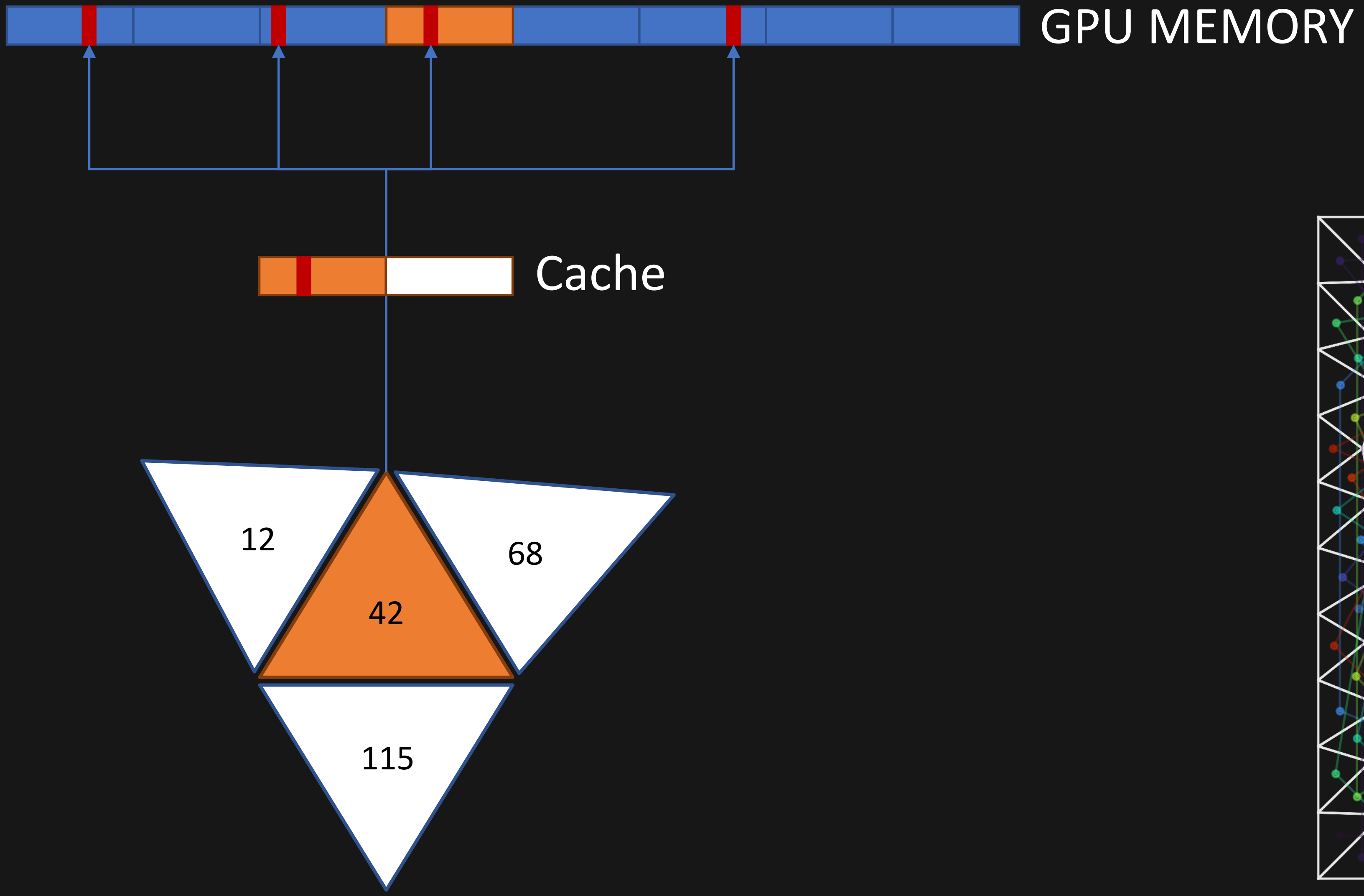
15%

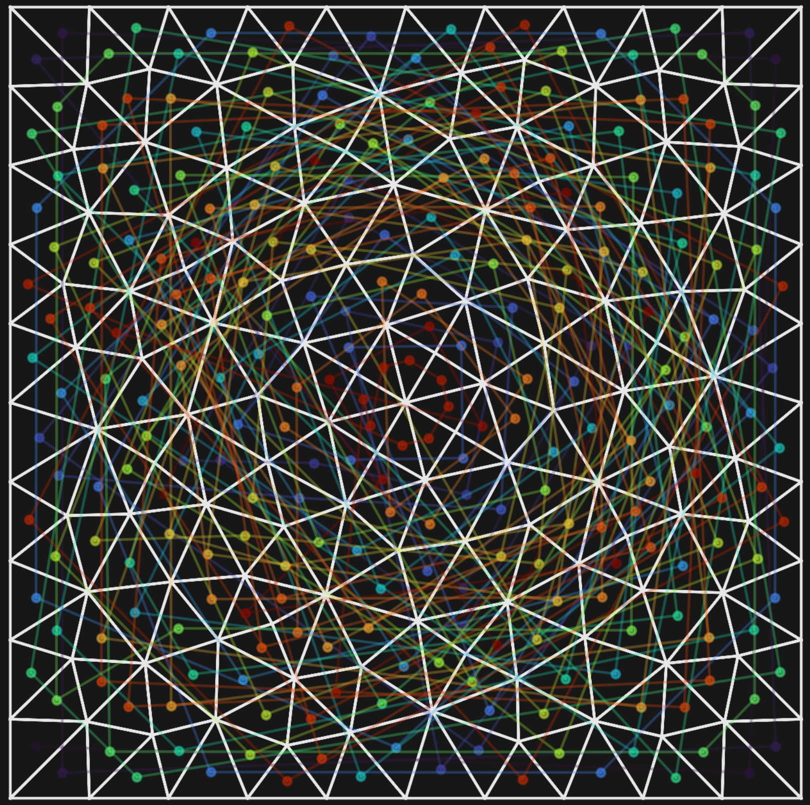
2 – Importance of memory locality



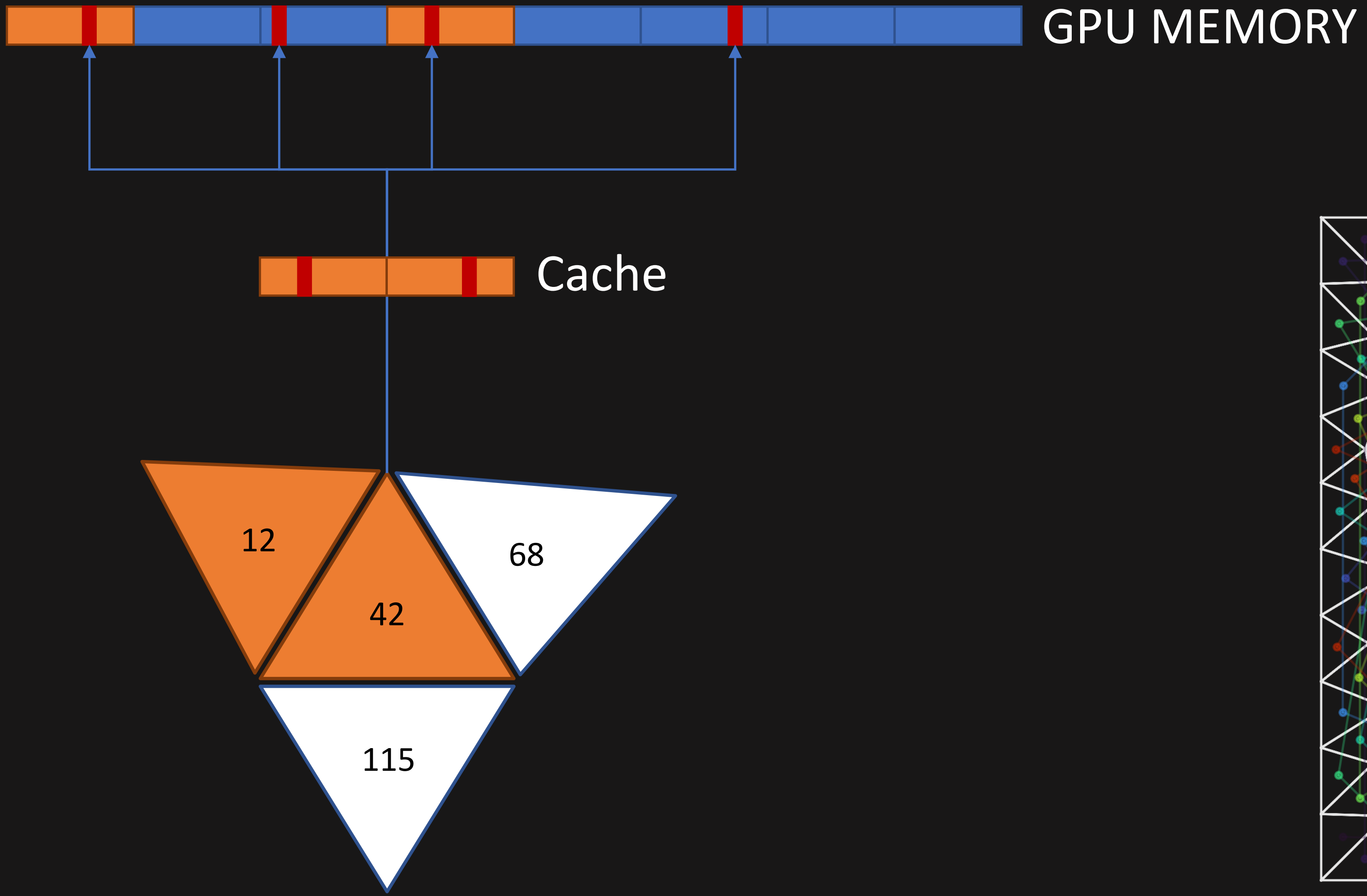


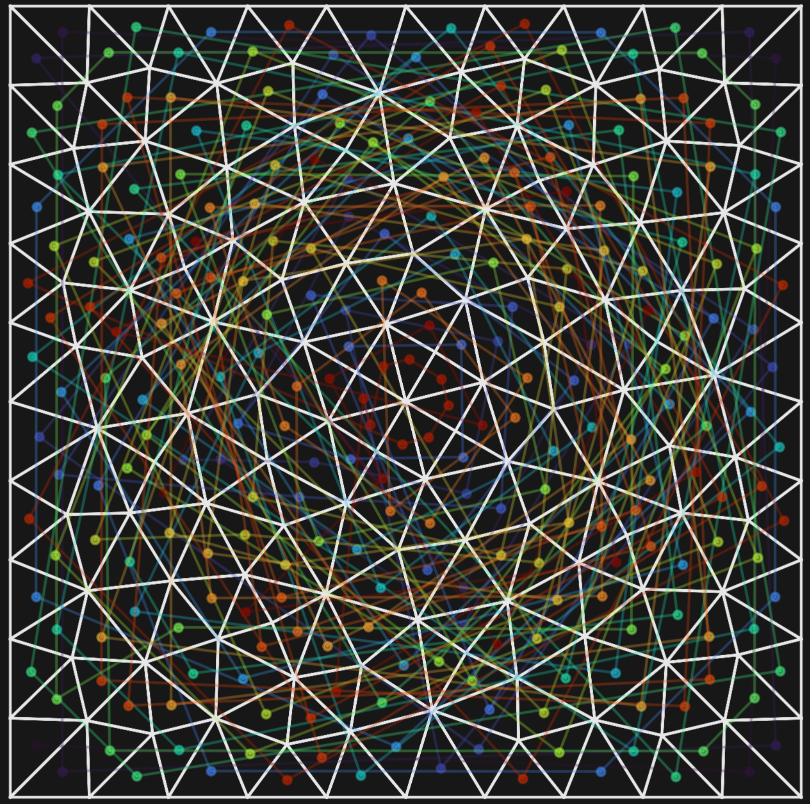
2 – Importance of memory locality



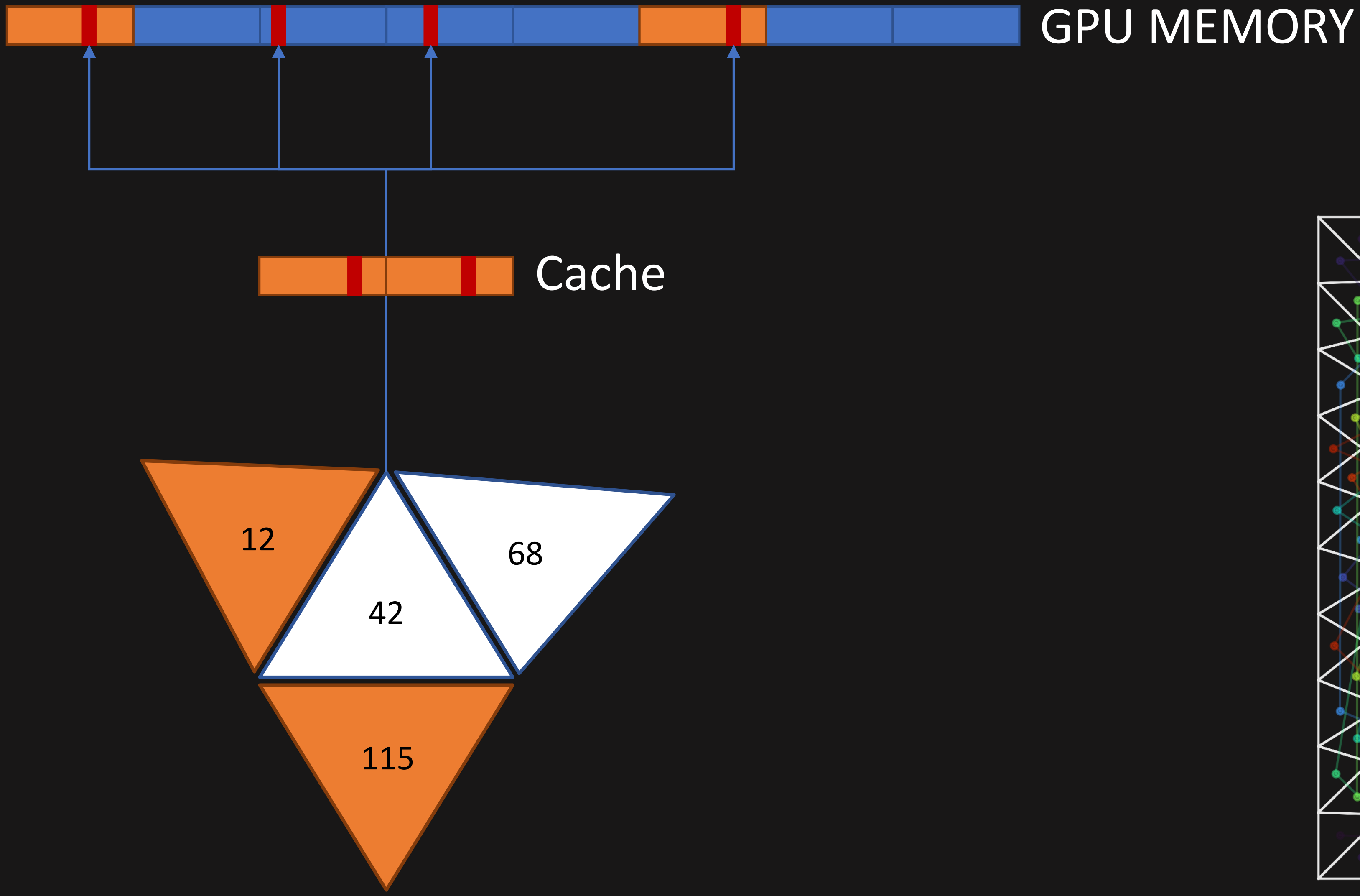


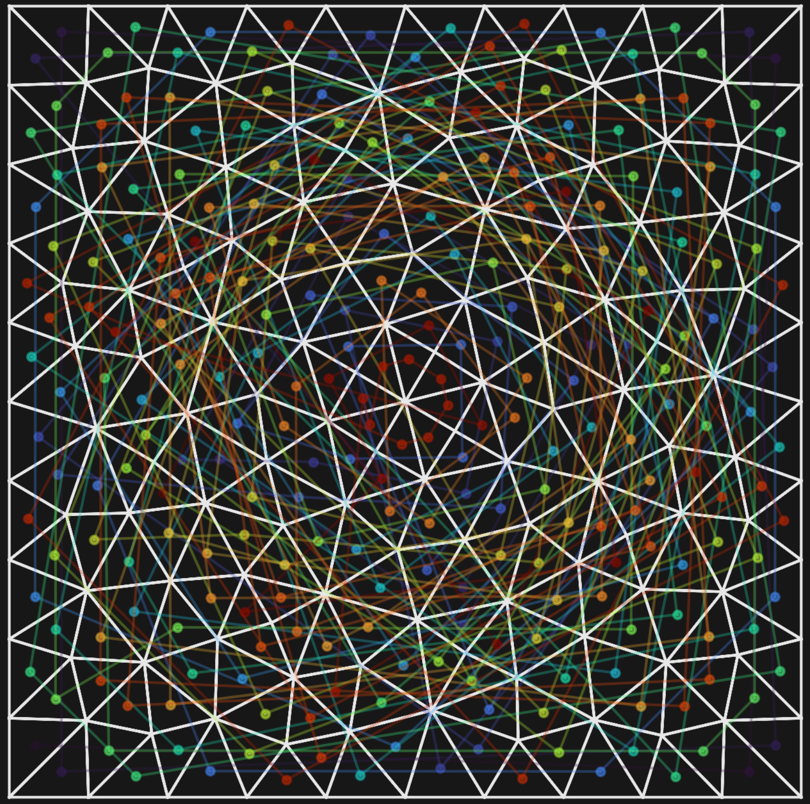
2 – Importance of memory locality



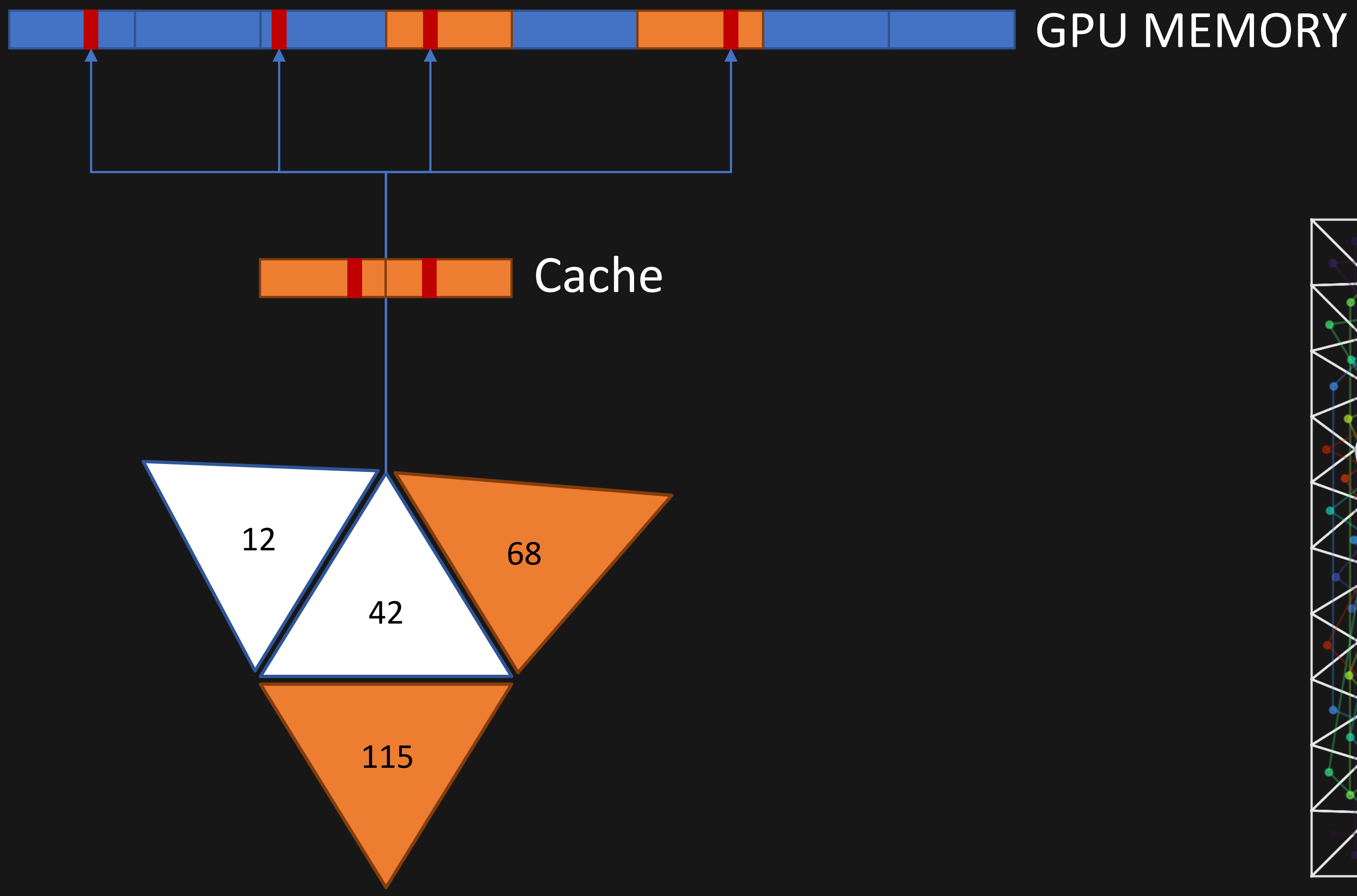


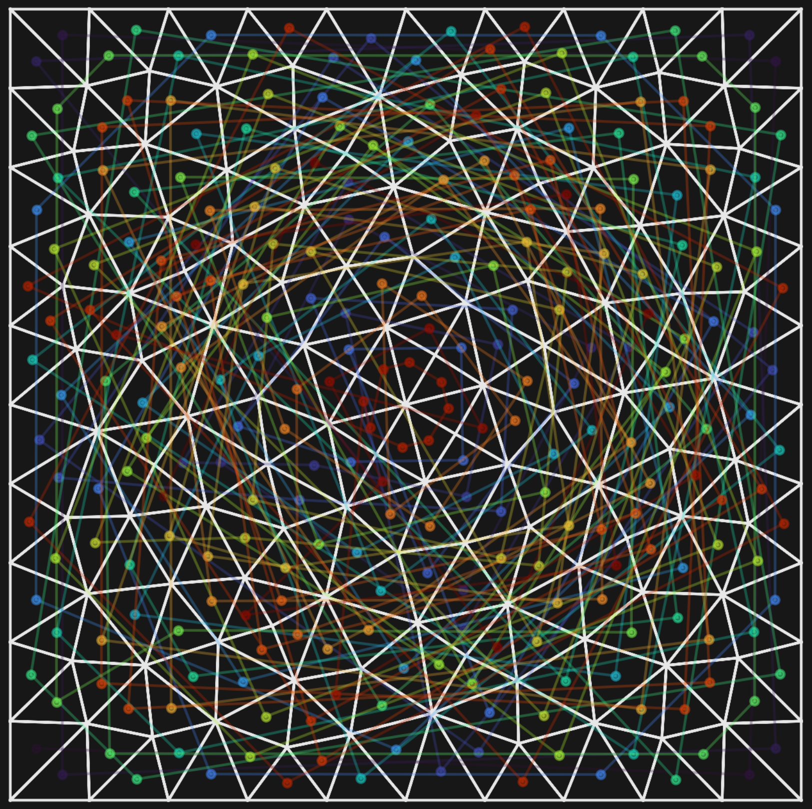
2 – Importance of memory locality



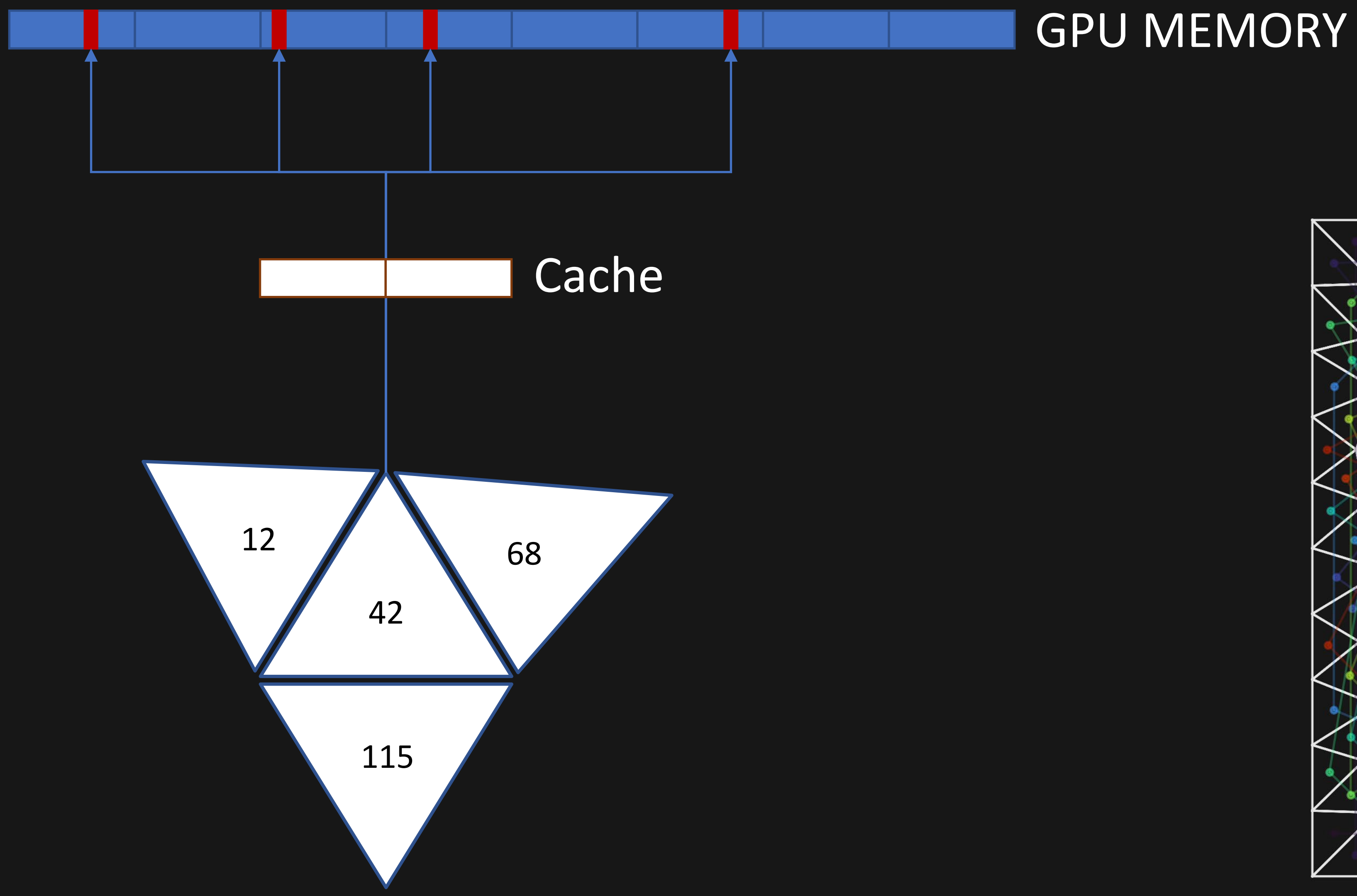


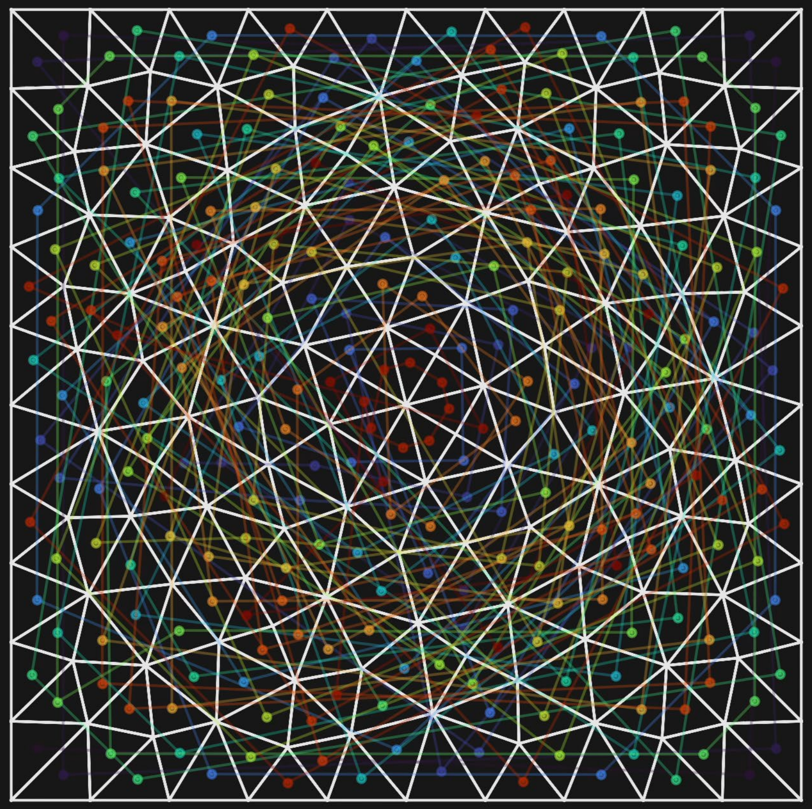
2 – Importance of memory locality



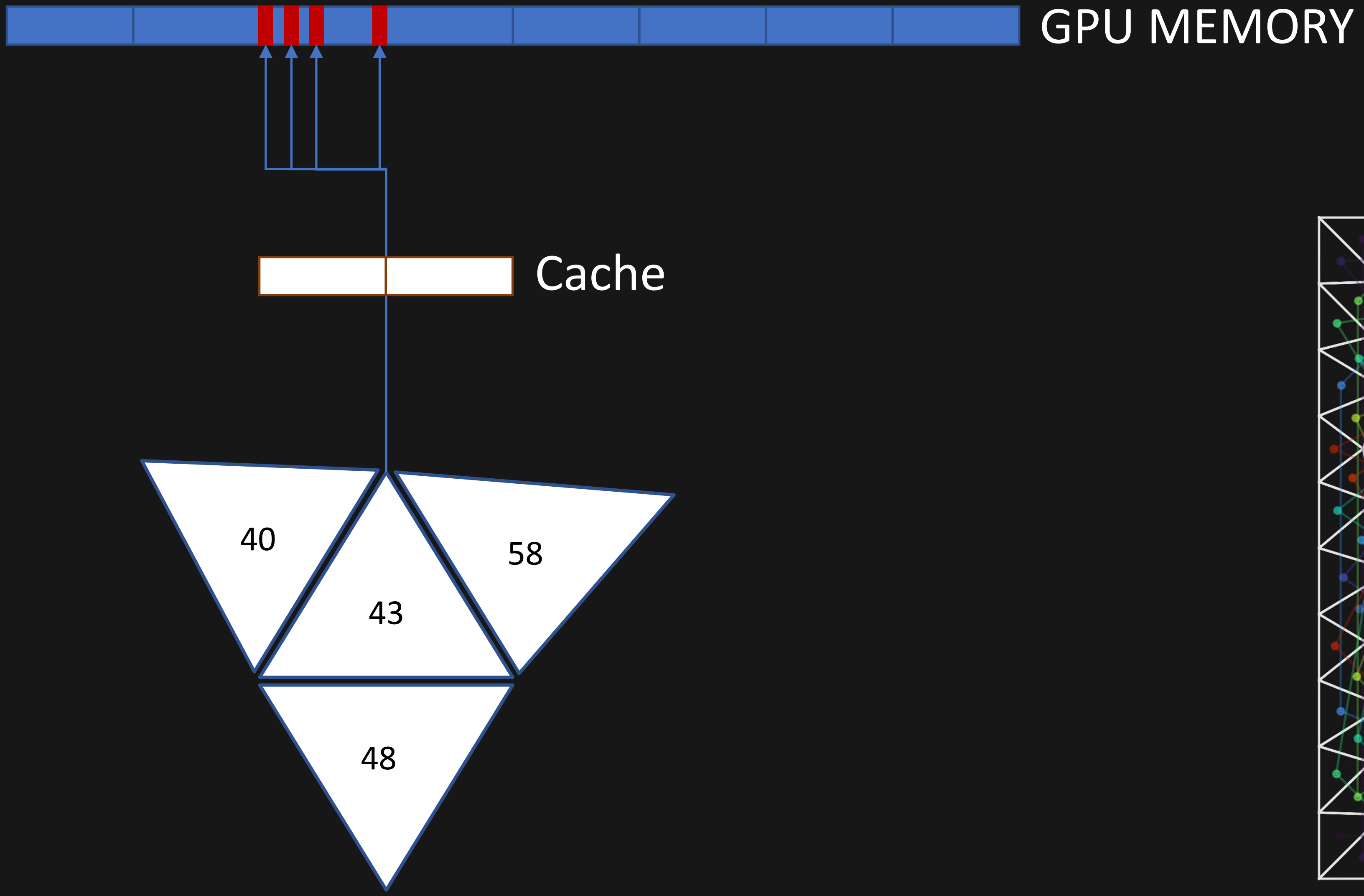


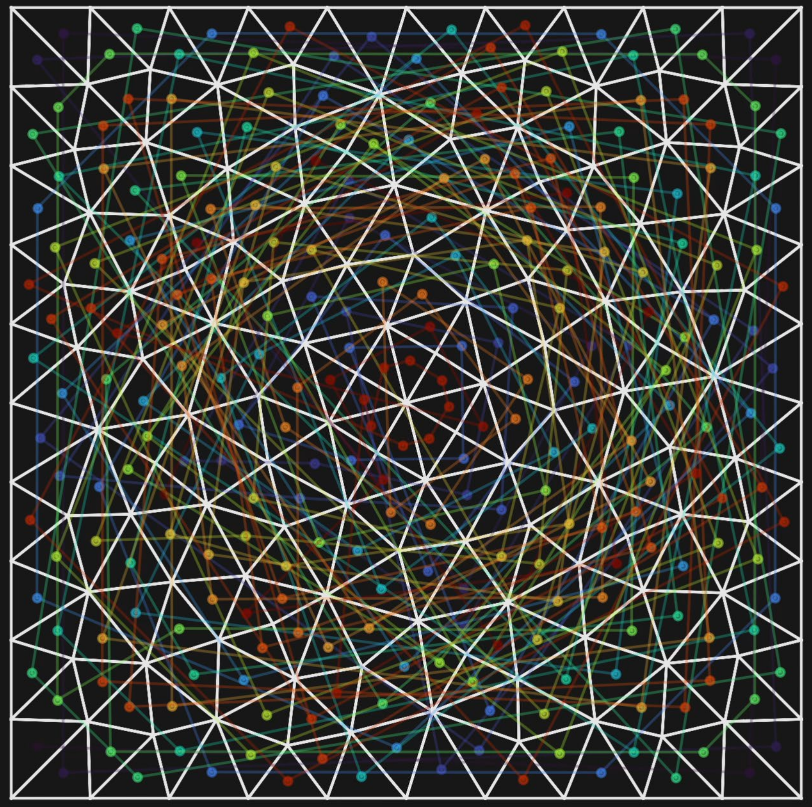
2 – Importance of memory locality



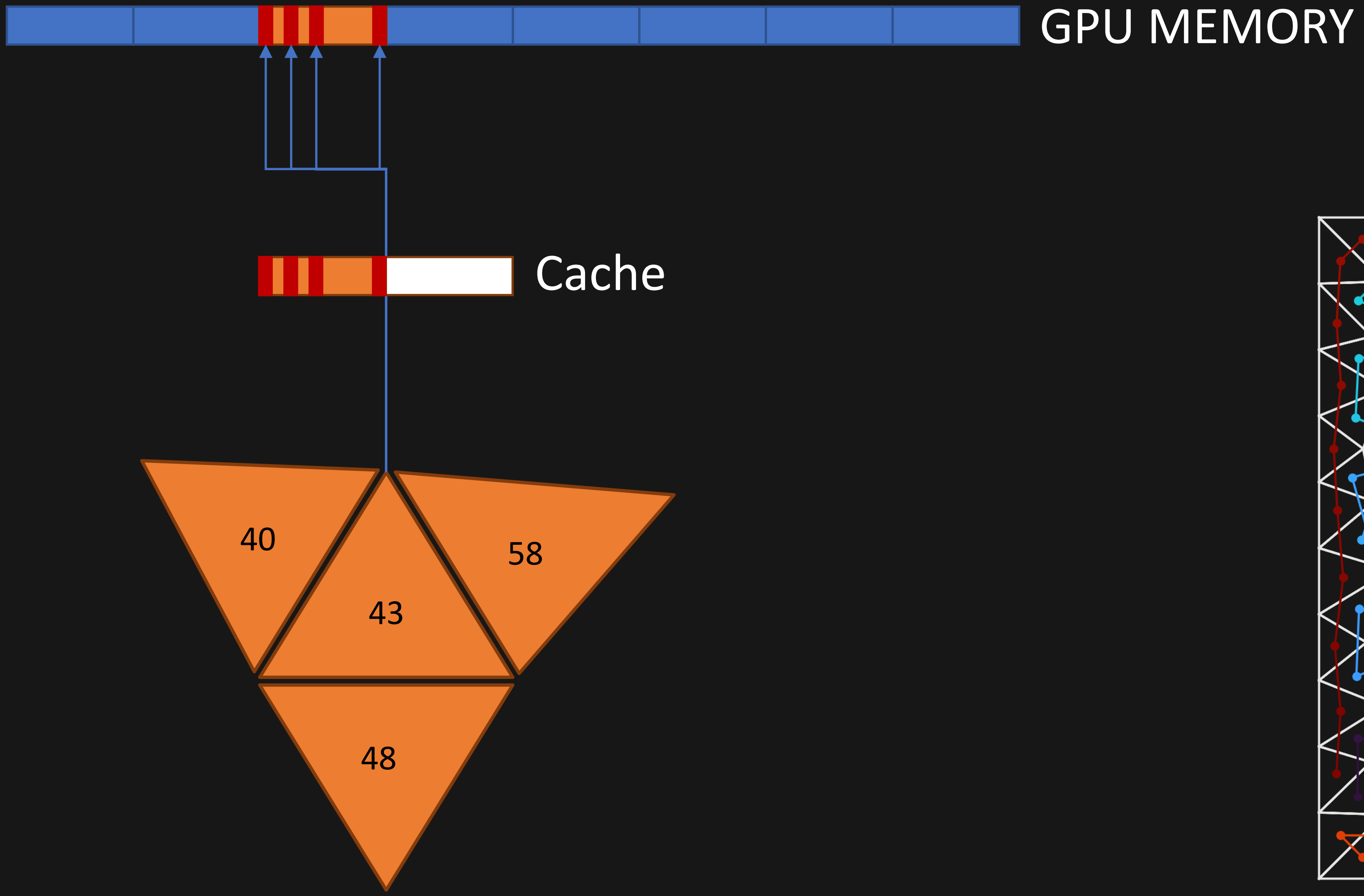


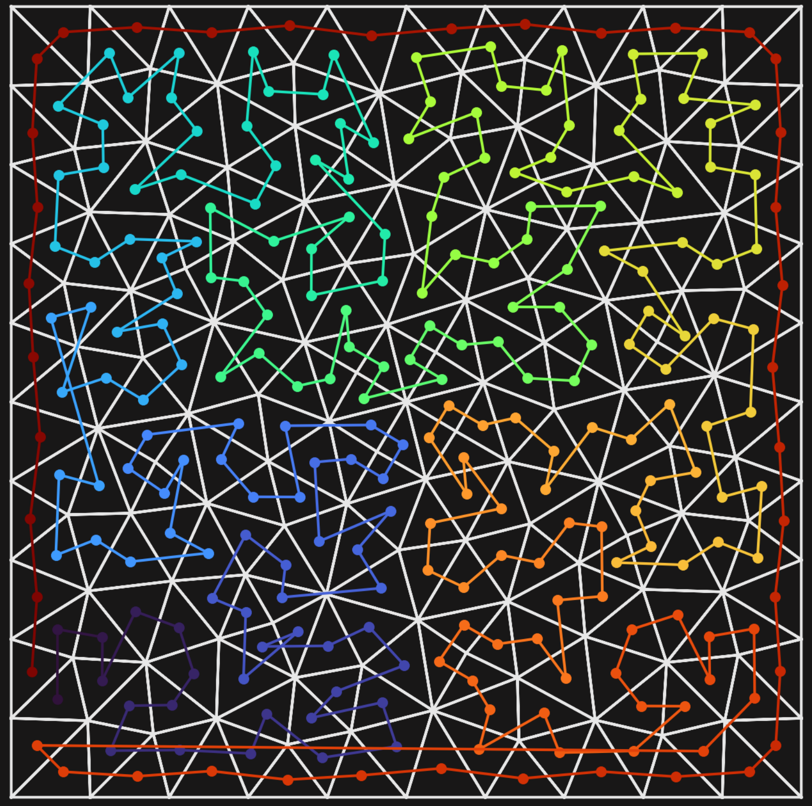
2 – Importance of memory locality





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.

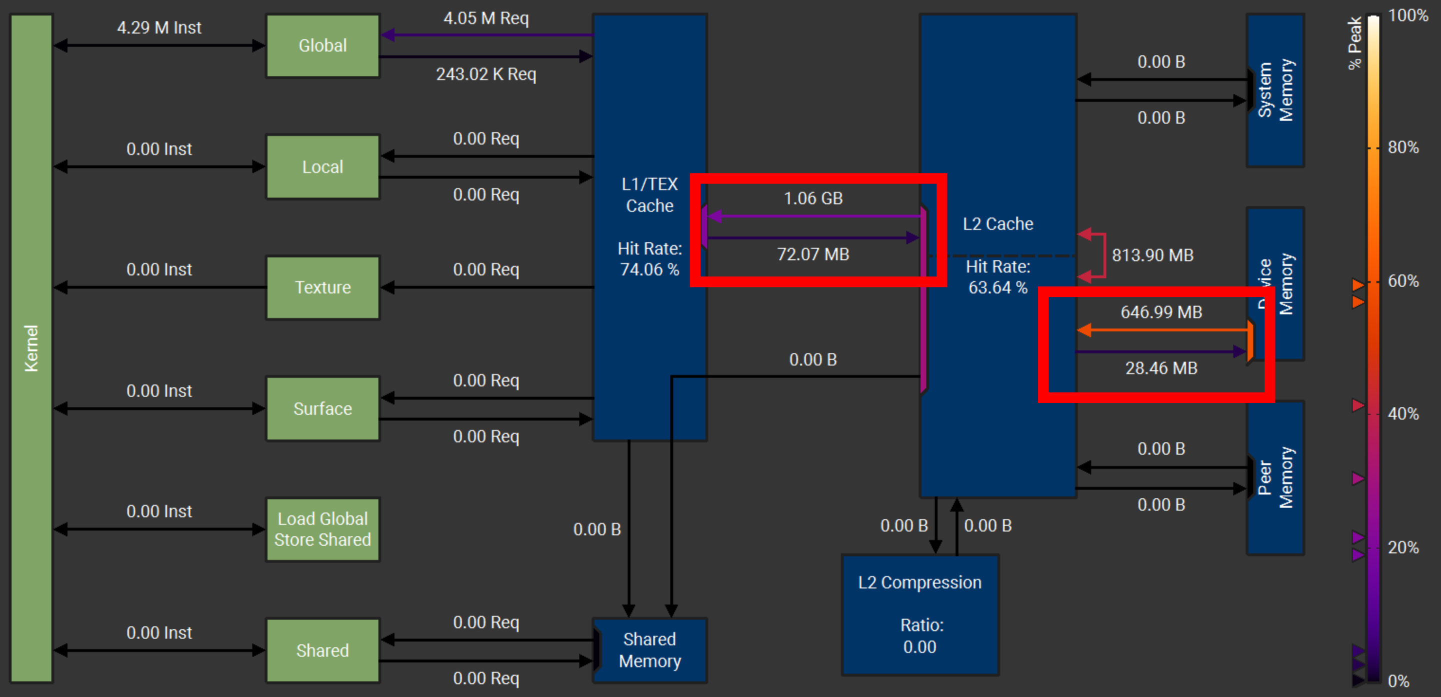
DRAM Excessive Read Sectors The memory access pattern for loads from device memory causes 21,733,744 sectors to be read from DRAM, which is 1.8x of the 12,160,697 sectors causing a miss in the L2 cache. The DRAM fetch granularity for read misses in L2 is 64 bytes, i.e. the lower or upper half of an L2 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

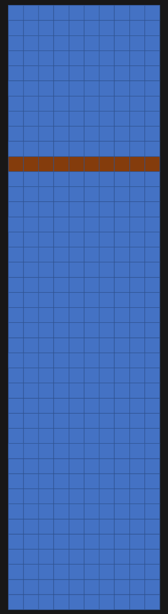
Memory Chart

Show As: Transfer Size

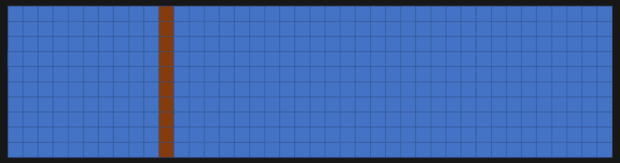


3 – So... let's optimize for locality, right ?

```
__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

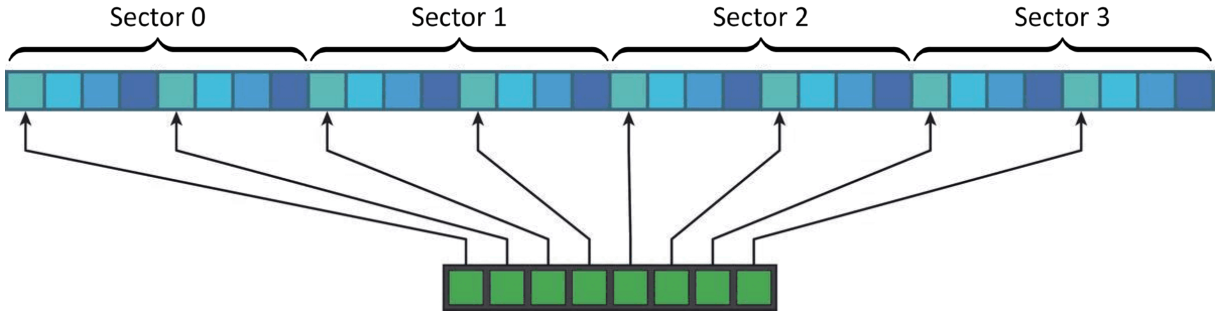
```
__global__ void mysum(const float* a, const float* b, float* c, int nf, int n){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n){
        for(int field = 0; field < nf; field++){
            // maximal locality
            c[tid * nf + field] = a[tid * nf + field] + b[tid * nf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

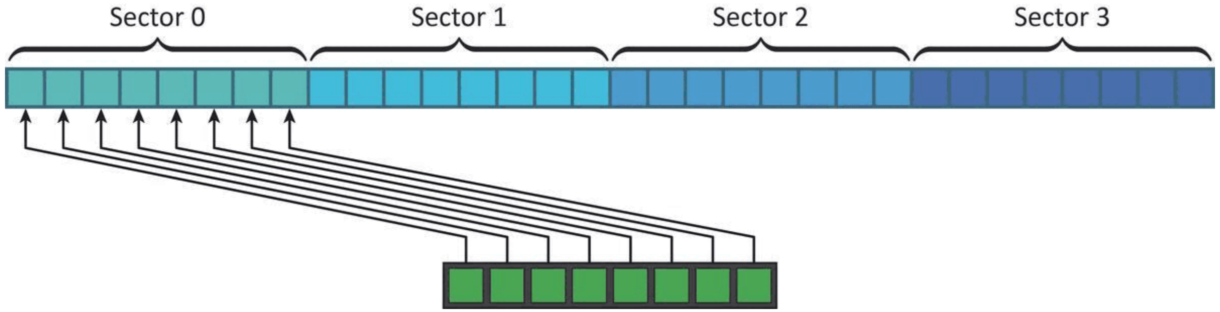
634 µs

```
__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 µs

3 – No, we need to consider coalescence



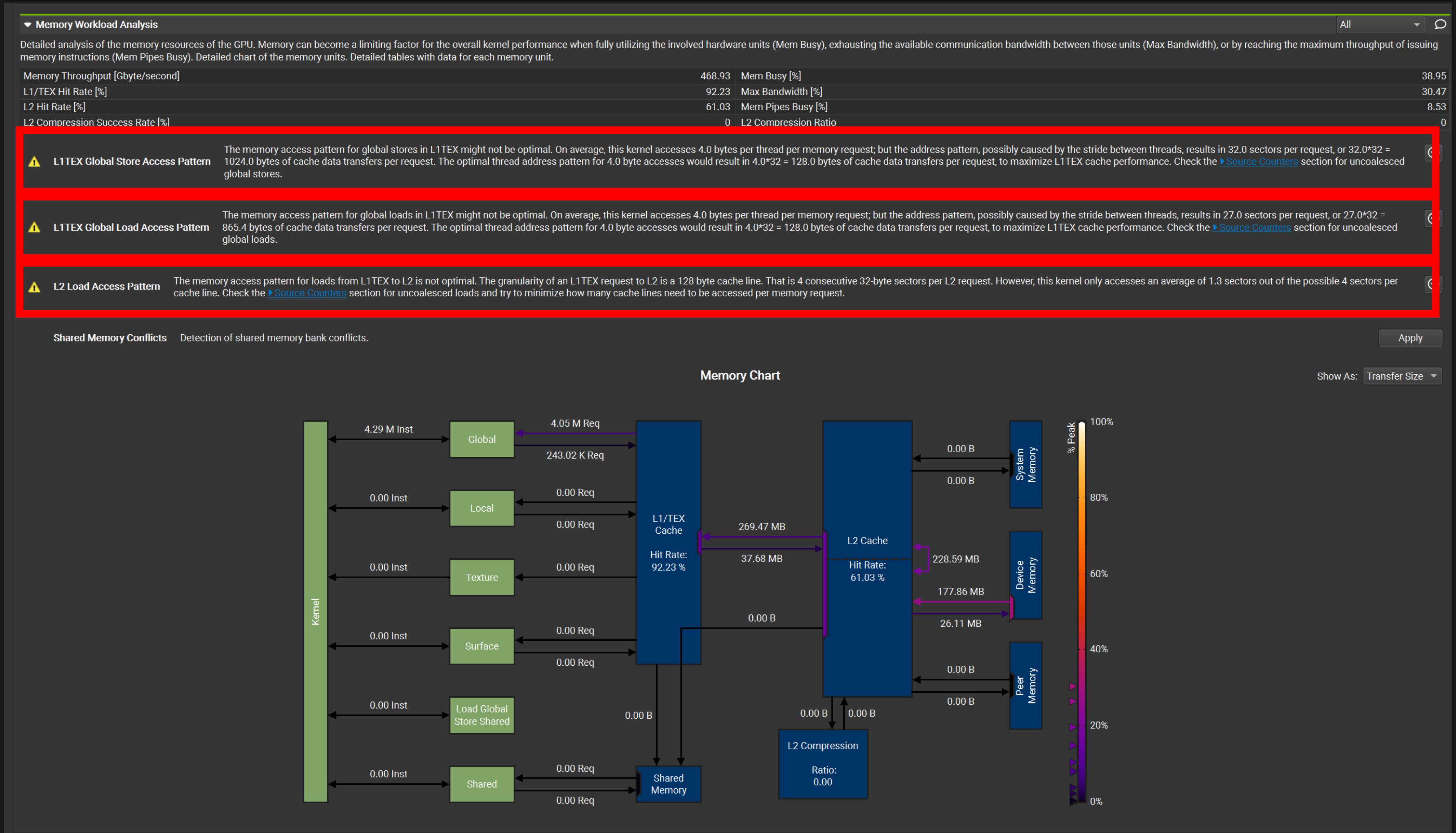



```
__global__ void mysum(const float* a, const float* b, float* c, int nf, int n){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n){
        for(int field = 0; field < nf; field++){
                  // maximal locality
            c[tid * nf + field] = a[tid * nf + field] + b[tid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



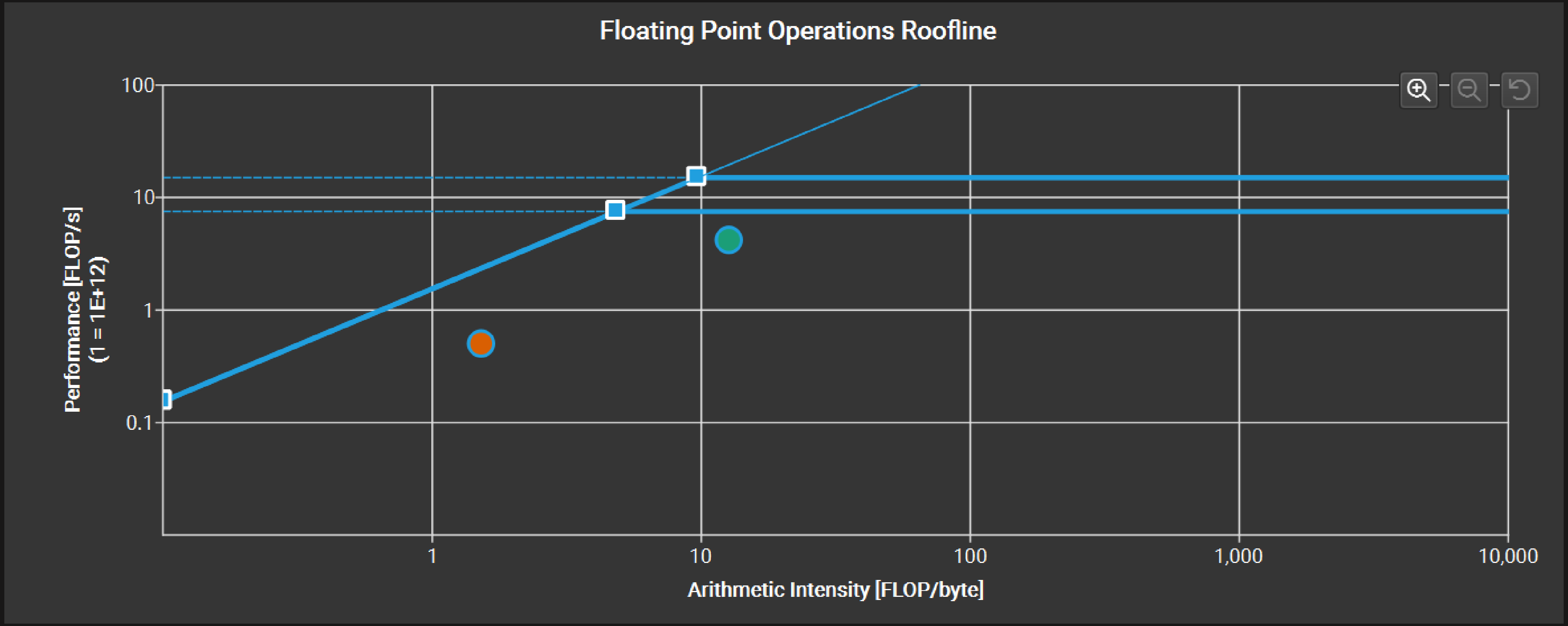
4 – Beware of doubles

```
__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

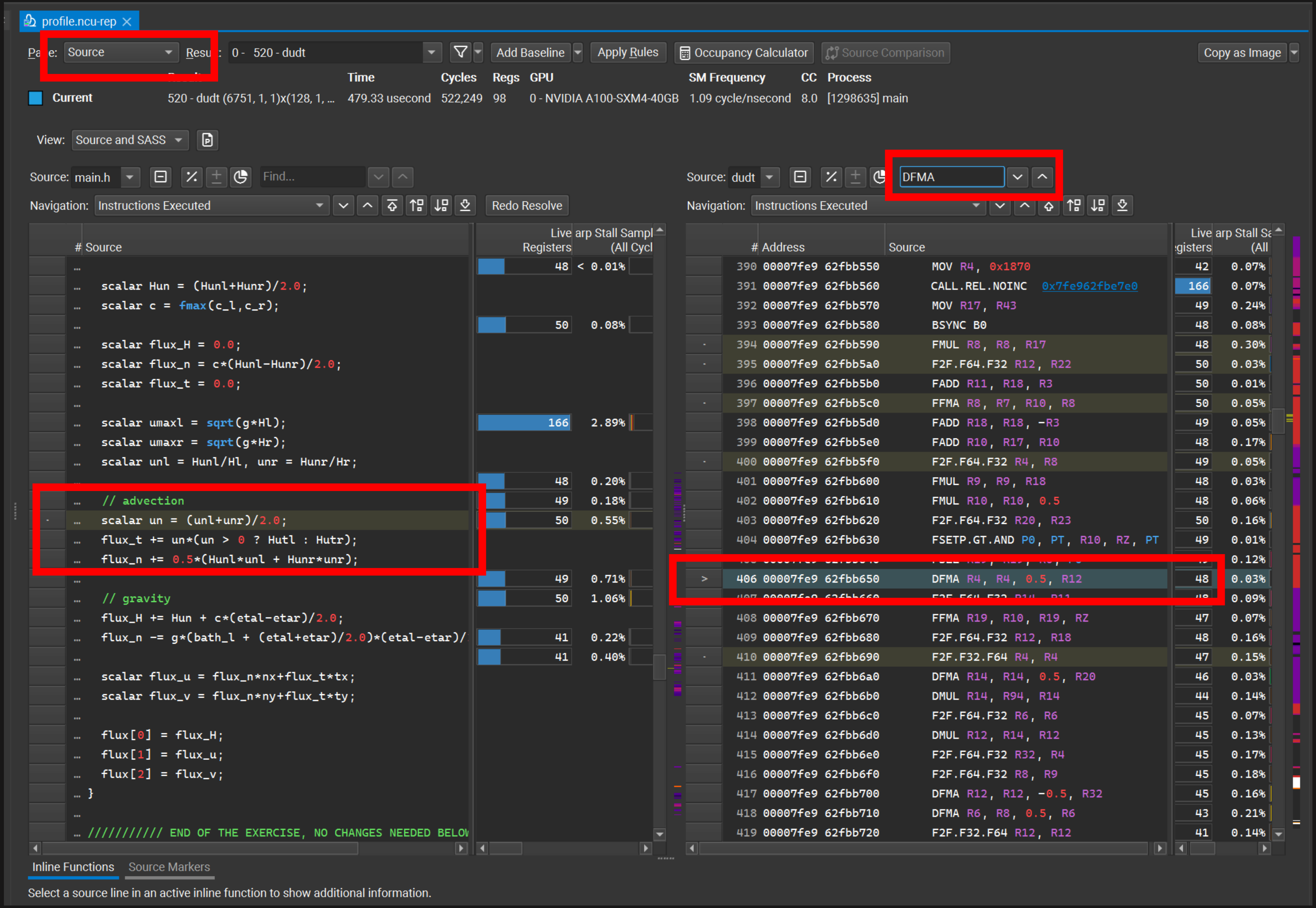
```
__device__ float inv2x2(const float mat[2][2], float
inv[2][2])
{
  float det = det2x2(mat);
  float ud = 1.0f / det;
  inv[0][0] =  mat[1][1] * ud;
  inv[1][0] = -mat[1][0] * ud;
  inv[0][1] = -mat[0][1] * ud;
  inv[1][1] =  mat[0][0] * ud;
  return det;
}
```

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`



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

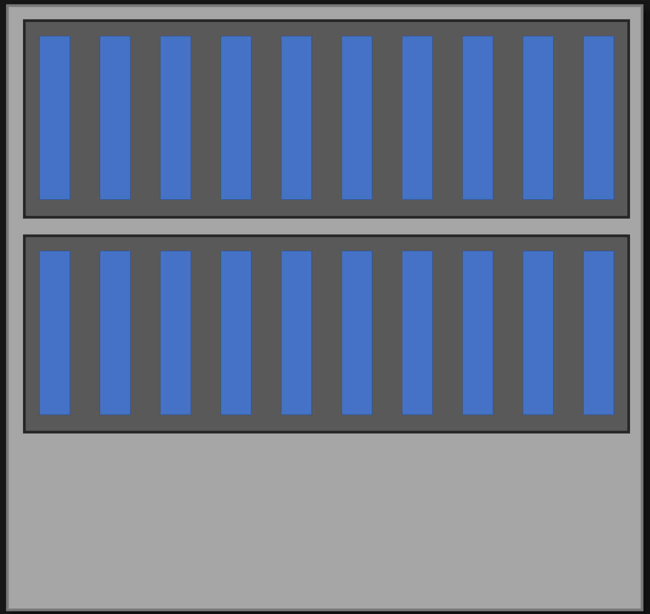
```
__device__ float inv2x2(const float mat[2][2], float
inv[2][2])
{
  float det = det2x2(mat);
  if(det){
    float ud = 1.0 / det;
    inv[0][0] =  mat[1][1] * ud;
    inv[1][0] = -mat[1][0] * ud;
    inv[0][1] = -mat[0][1] * ud;
    inv[1][1] =  mat[0][0] * ud;
  }
  else{
    printf("Singular matrix 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;
}
```

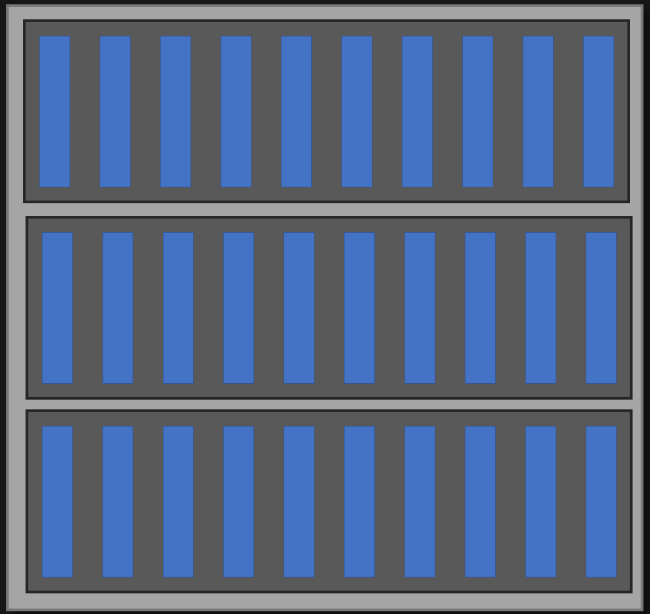
5 – Watch the occupancy!

```
__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!

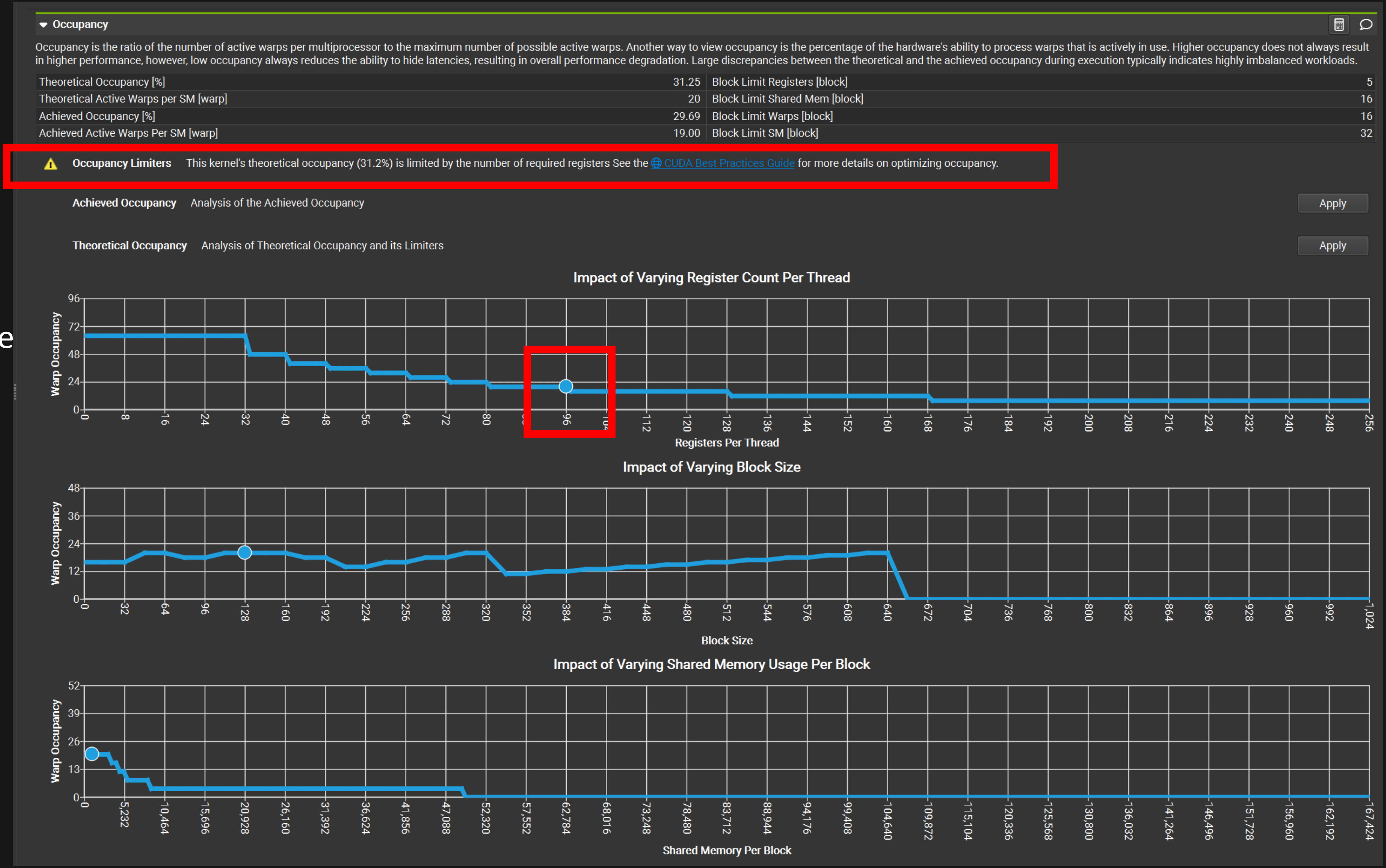
```
__device__ float inv2x2(const float mat[2][2], float
inv[2][2])
{
  float det = det2x2(mat);
  float ud = 1.0 / det;
  inv[0][0] =  mat[1][1] * ud;
  inv[1][0] = -mat[1][0] * ud;
  inv[0][1] = -mat[0][1] * ud;
  inv[1][1] =  mat[0][0] * ud;
  return det;
}
```



5 – Is the occupancy a limiting factor?

- ncu will tell you

- Often the case that occupancy is limited by registers, but to be « on a step »



Exercice 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

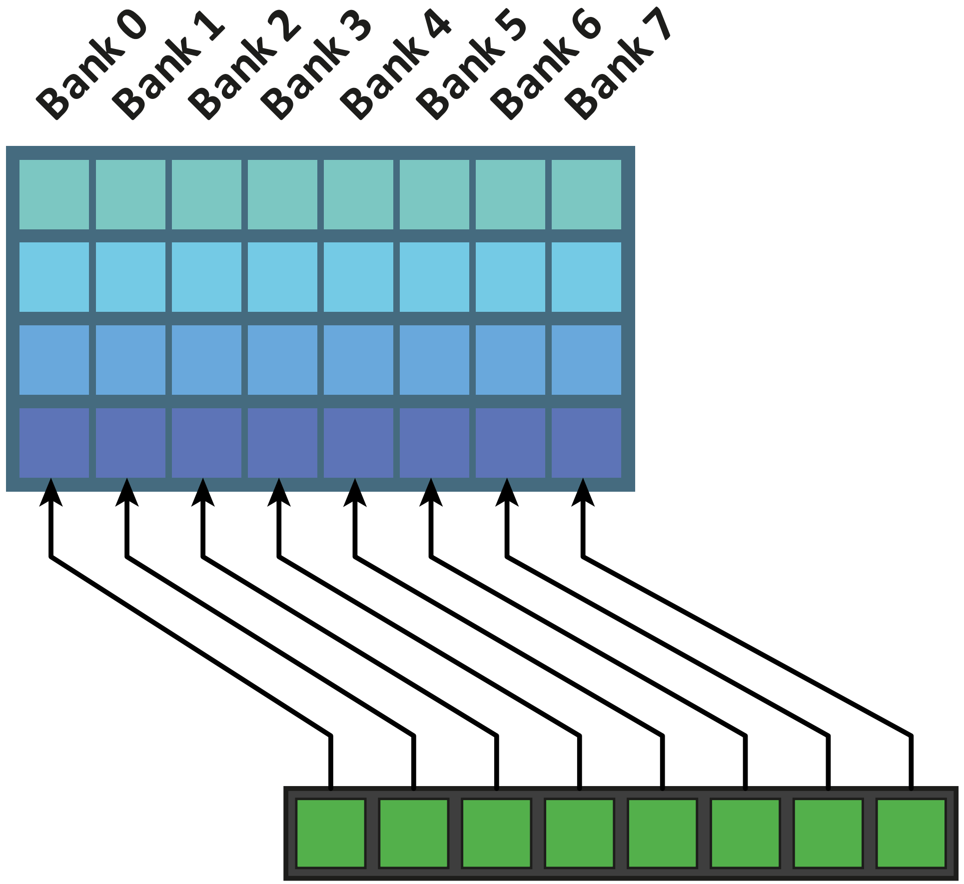
- Kernels comparable in duration
- Kernels share variables

➢ Makes sense to merge the kernels to avoid a read and write from global memory

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

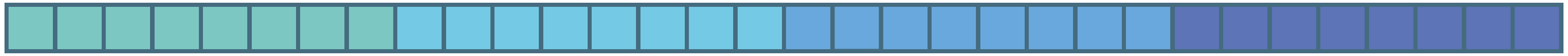


- Shared memory organzed in « Banks »
- Simultaneous accesses to the same bank are serialized
- Consecutive threads should access consecutive banks

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
- `--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, because

Some edged cases encountered

- __launch_bounds__() slowing down the code
- Optimizing for performance is complicated. So complicated that the compiler itself often gets confused

```
if ((tri_r >= 0) || (1)){ // there is someone on the right
    int prism_r = tri_r >= 0 ? tri_r*n_layers + front : -1;

if ((tri_r >= 0) || (1)){ // there is someone on the right
    int prism_r = tri_r*n_layers + front; // same result, but overall code is 1.33x faster
```

- Adding a big bloc of code made everything 1.2x faster, even when it was never accessed.

Conclusion :

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

You never know.