

GPU optimization techniques and tools

A collection of ideas to maybe improve your GPU performance

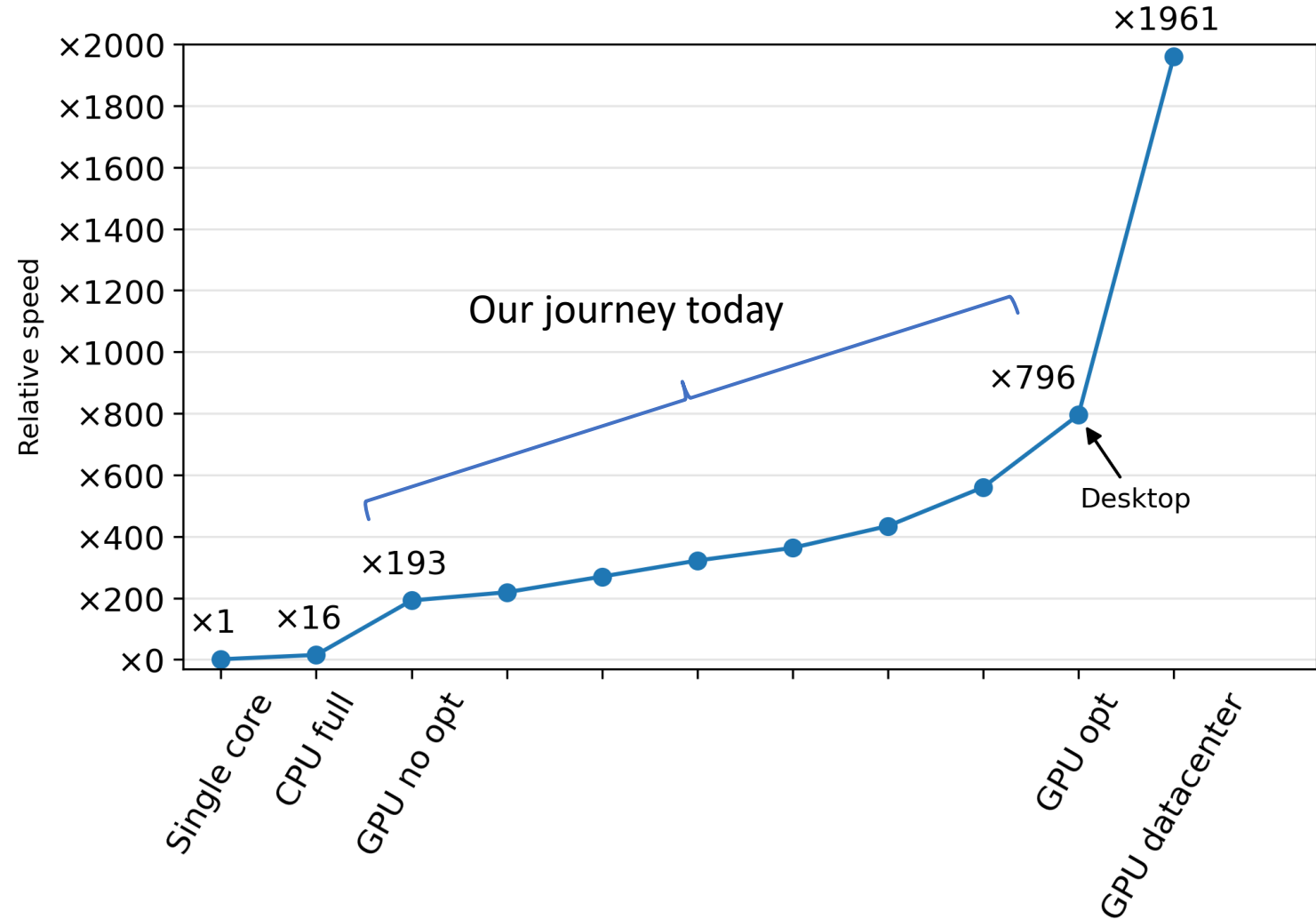
*Presented data acquired over a two years period of hands-on experimentation and accumulated frustration and suffering.

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



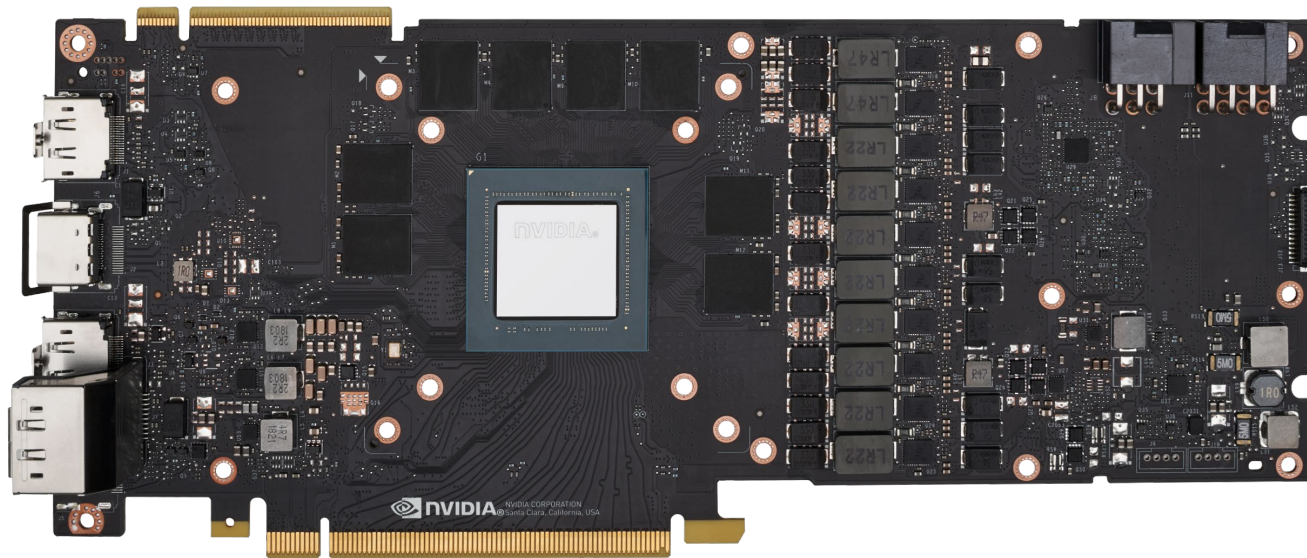
Plan for this session

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

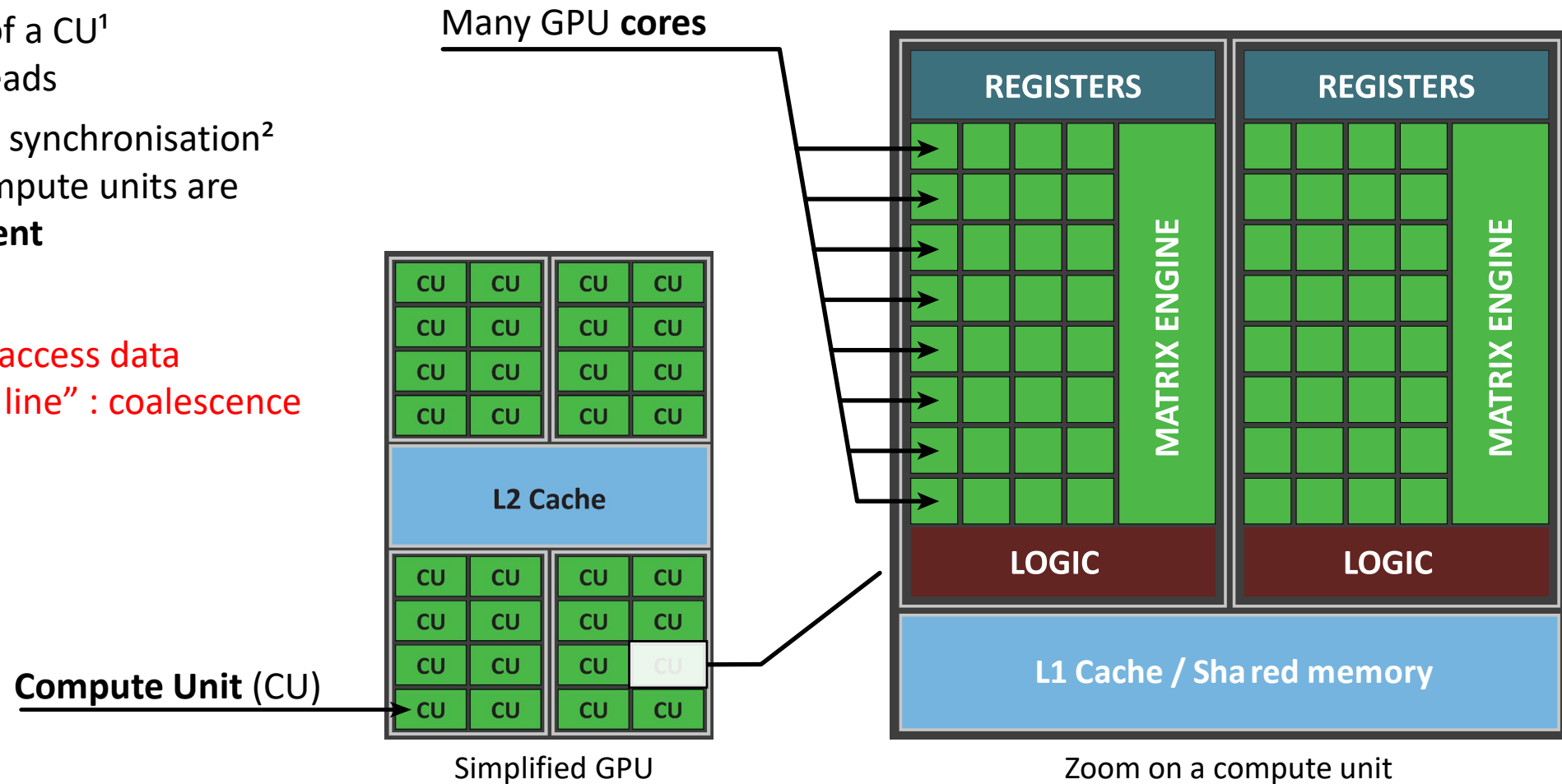
- SIMD-like execution model
- Coalescent memory access
- Very high memory latency
- Designed for higher arithmetic intensity
- Very limited cache per thread
- ...



What is a GPU : the execution model

Defining characteristics:

- Cores of a CU¹ are **not independent**
- Computations inside of a CU¹ is the same for all threads
- No data exchange and synchronisation² outside of the CU. Compute units are **completely independent**
- Cores in a CU want to access data from the same “cache line” : coalescence



Nomenclature

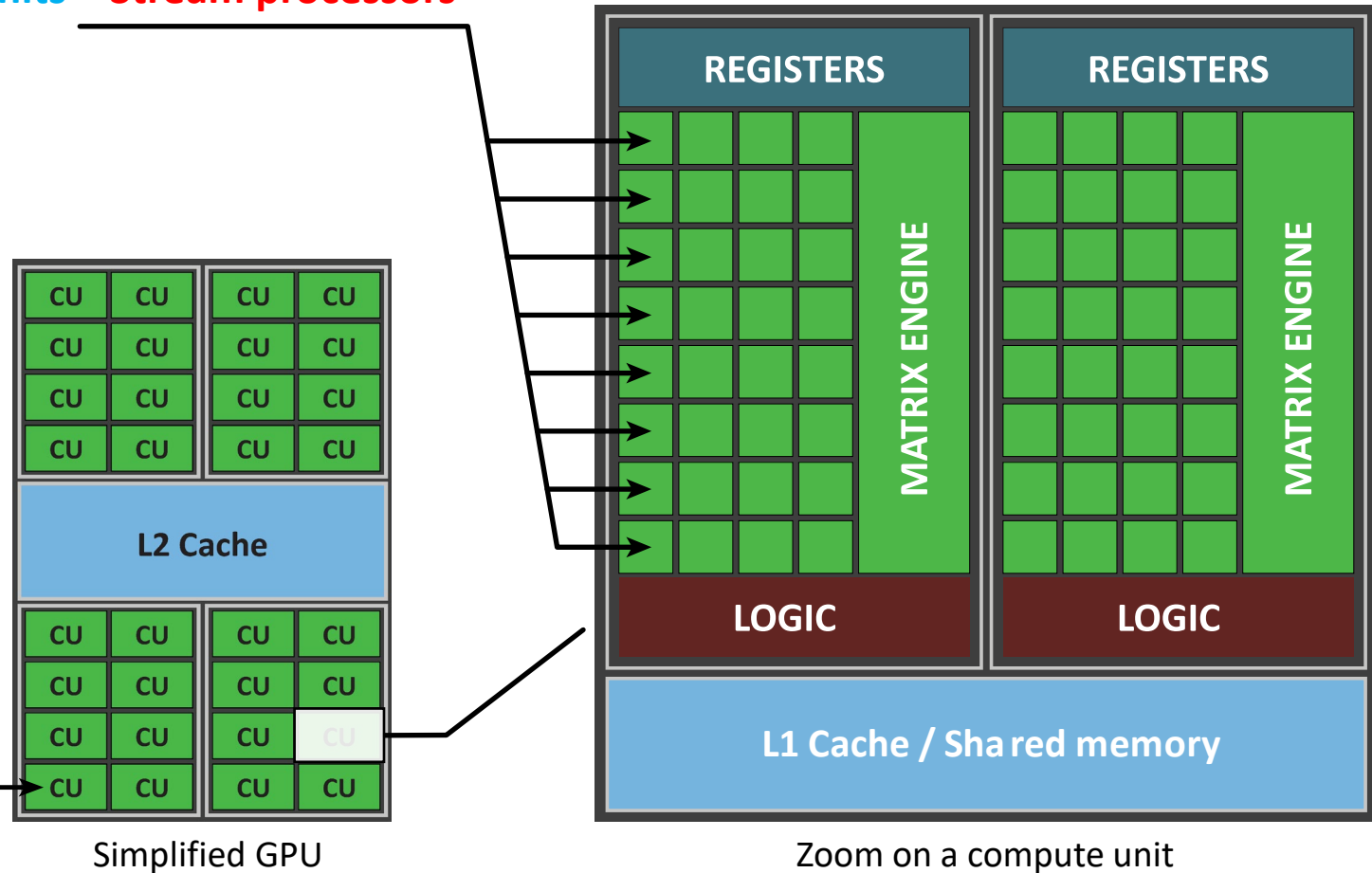
Green = NVIDIA

Blue = INTEL

Red = AMD

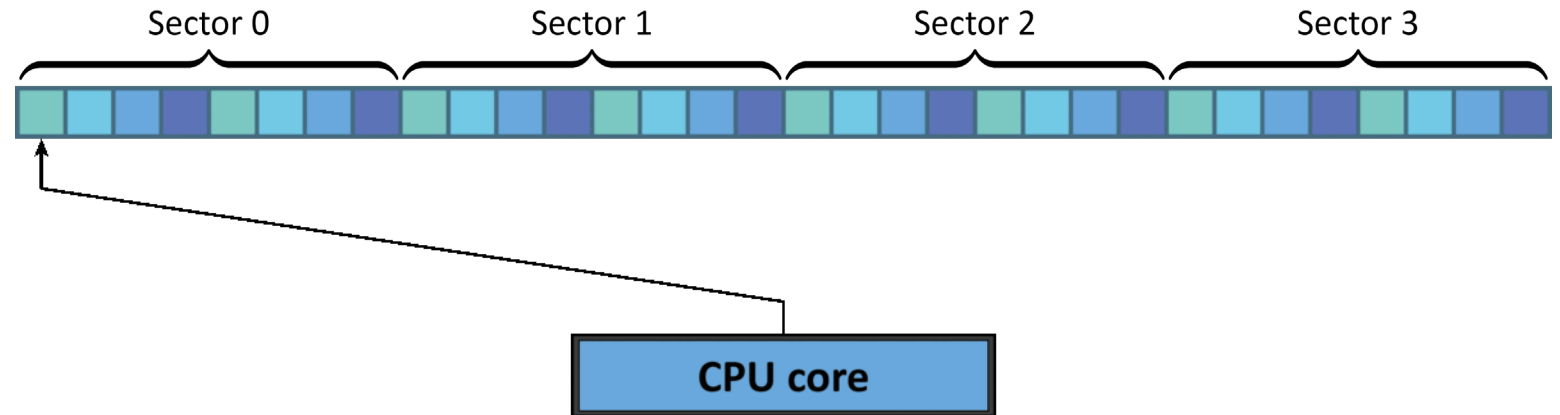
(Cuda) cores = Shading Units = Stream processors

Streaming multiprocessor = Core = Compute Unit



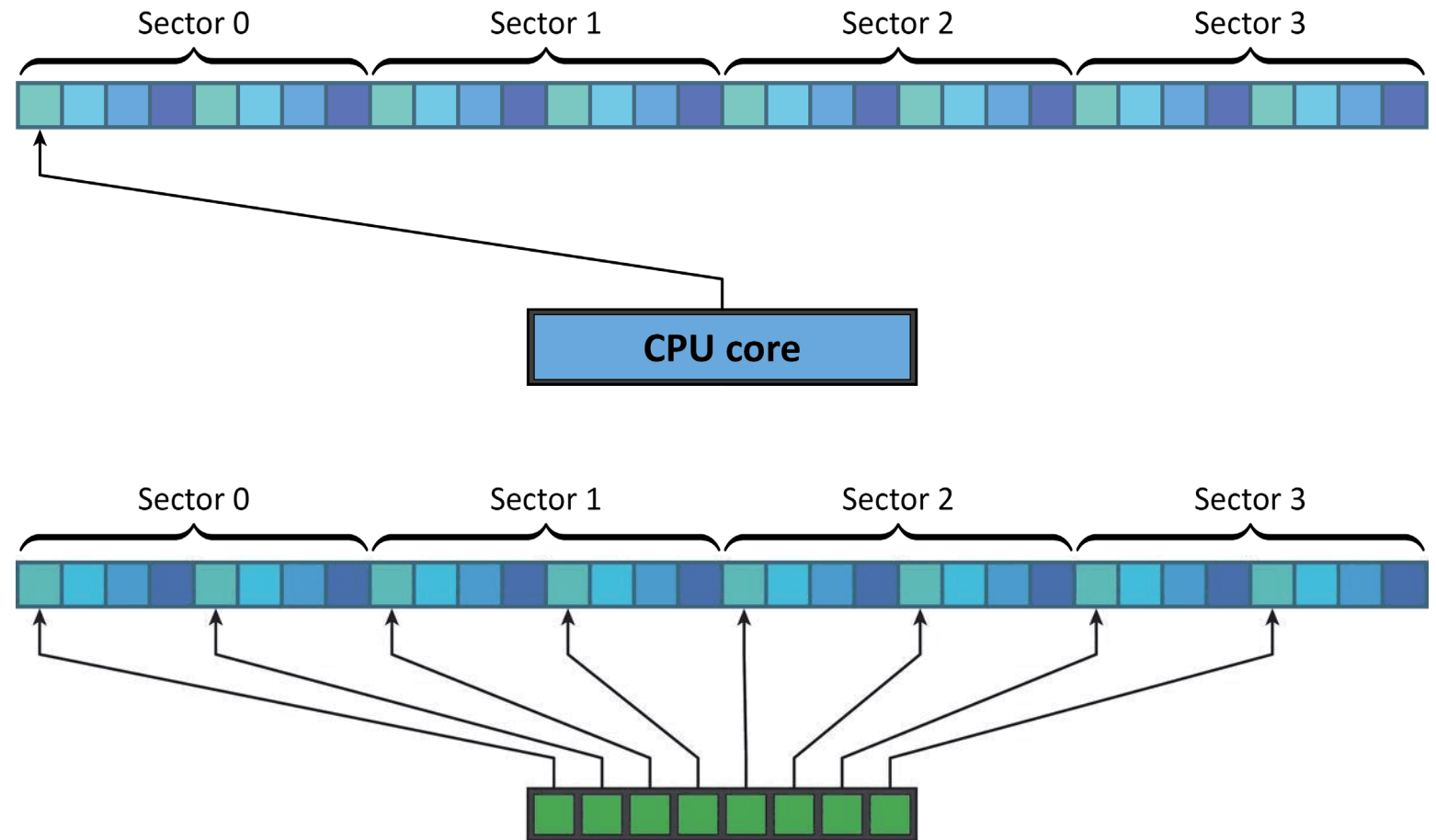
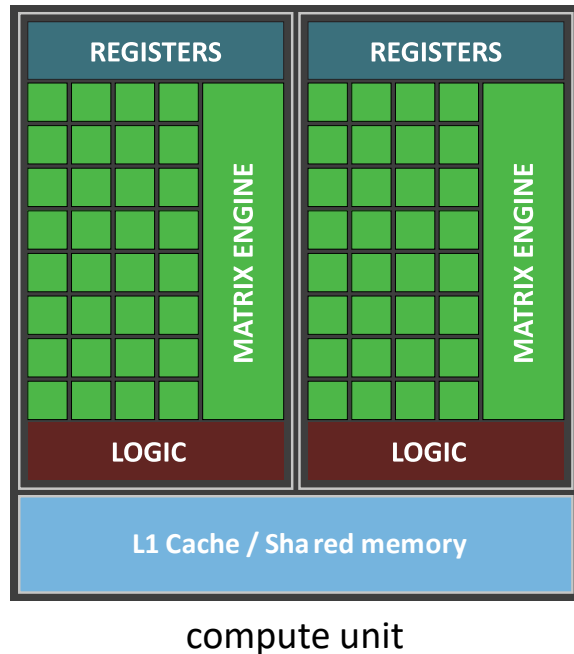
Coalescent memory access

- On the CPU : we want to maximize locality



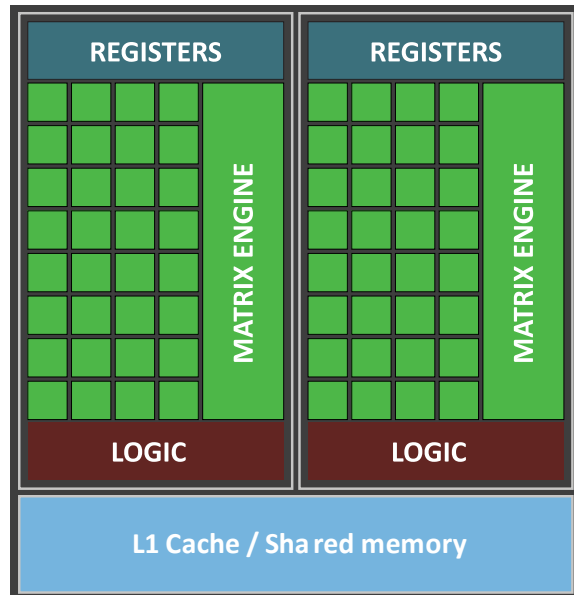
Coalescent memory access

- 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

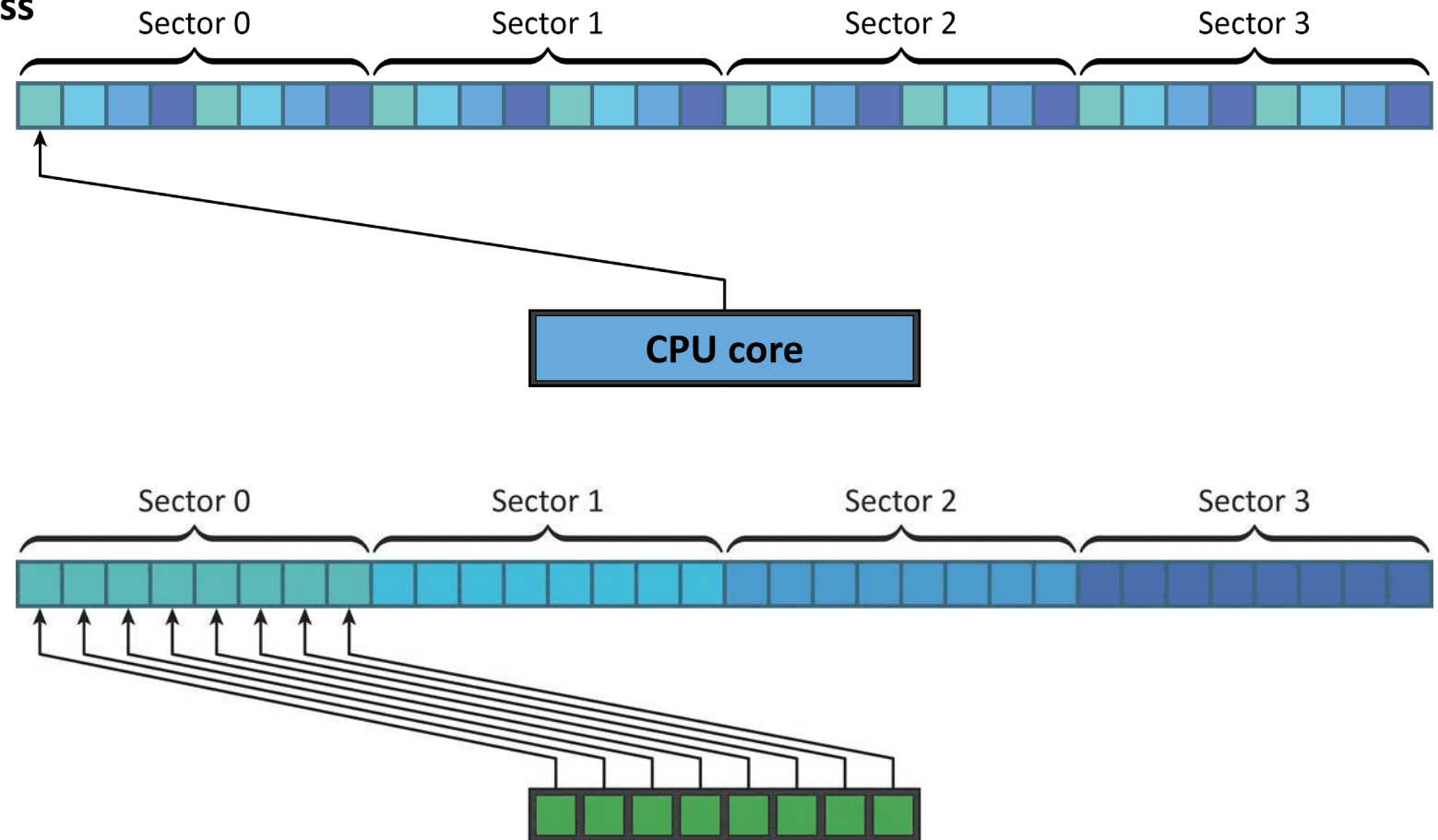


Coalescent memory access

- 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



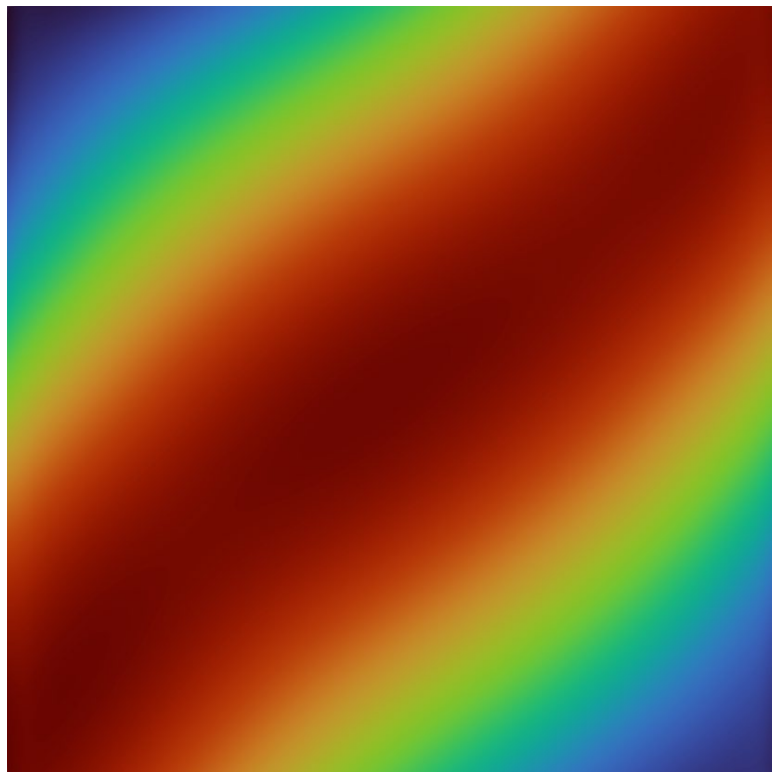
1 – Double vs float : which one should you choose, and why is it float*?

- RTX3080:

- Float : 29.77 TFLOPS
- Double : 0.47 TFLOPS

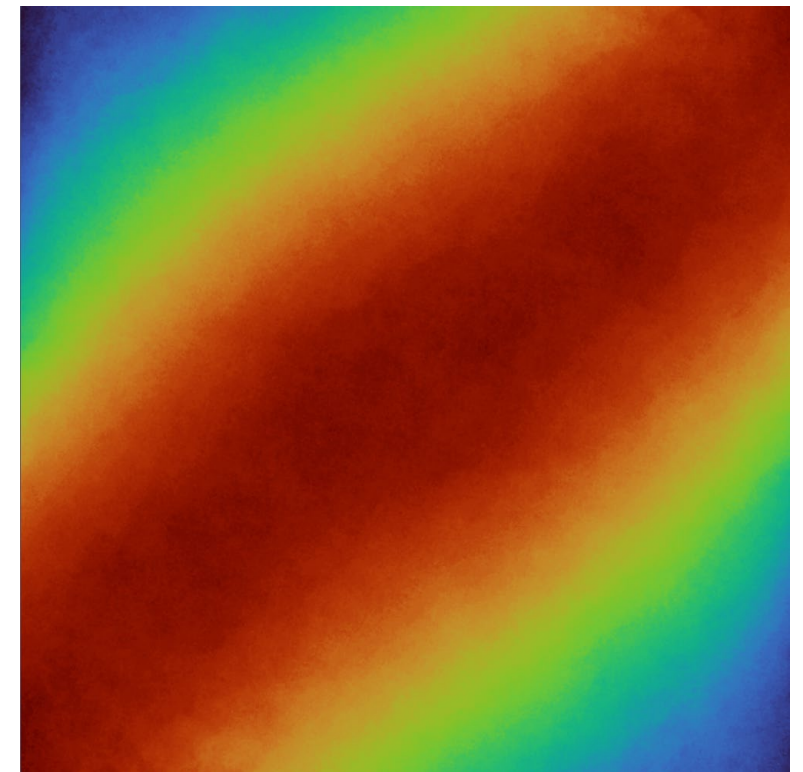
- A100:

- 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++){
```

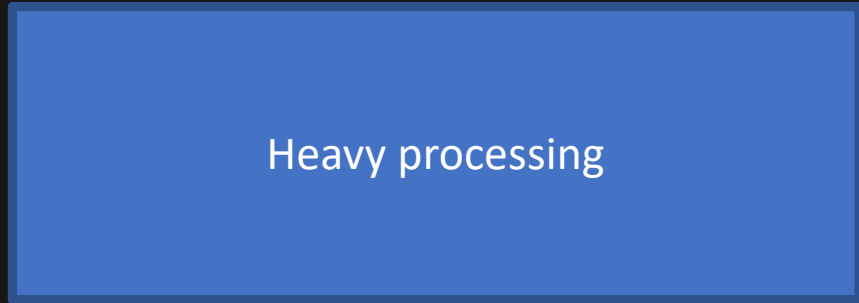
```
        // loading data
```

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

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

} 4%

```
        // some heavy computation
```



} 95%

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

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

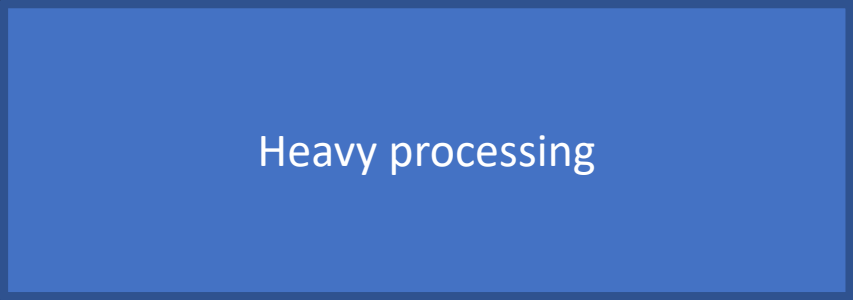
} 1%

```
    }
```

```
}
```

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;
    }
}
```

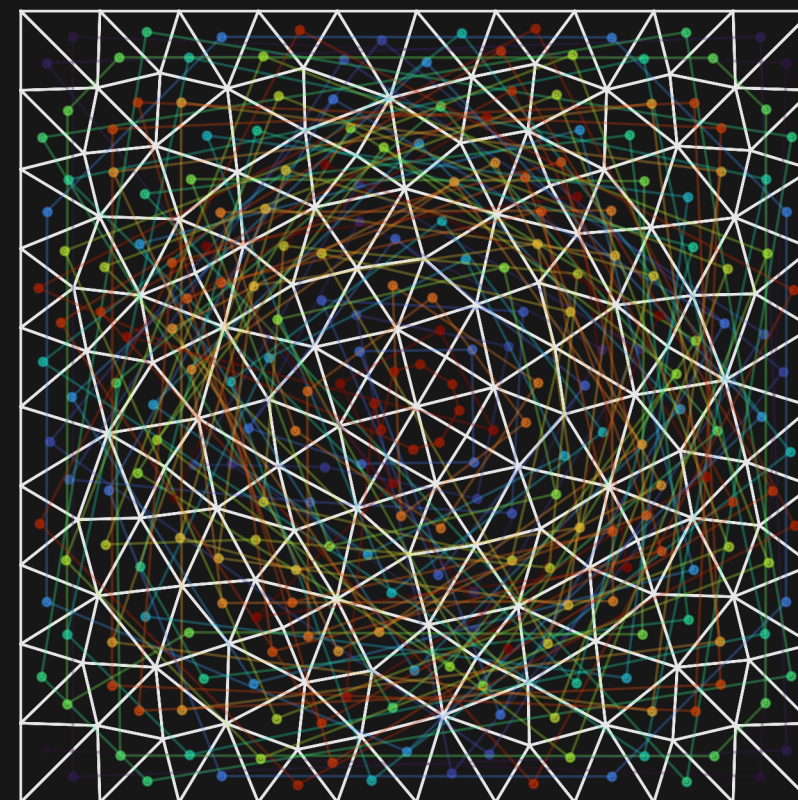
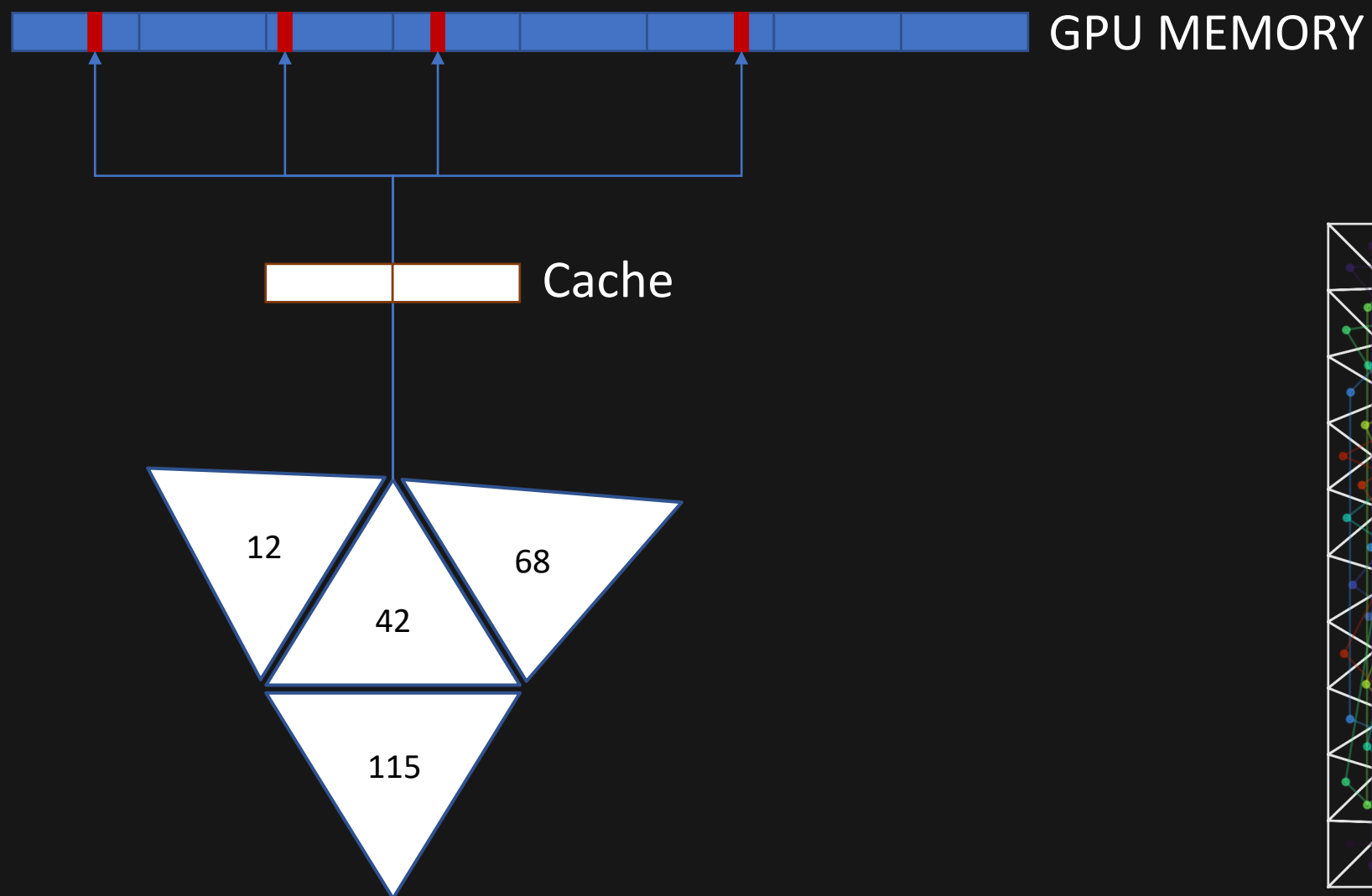
Memory is (almost always) the bottleneck

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

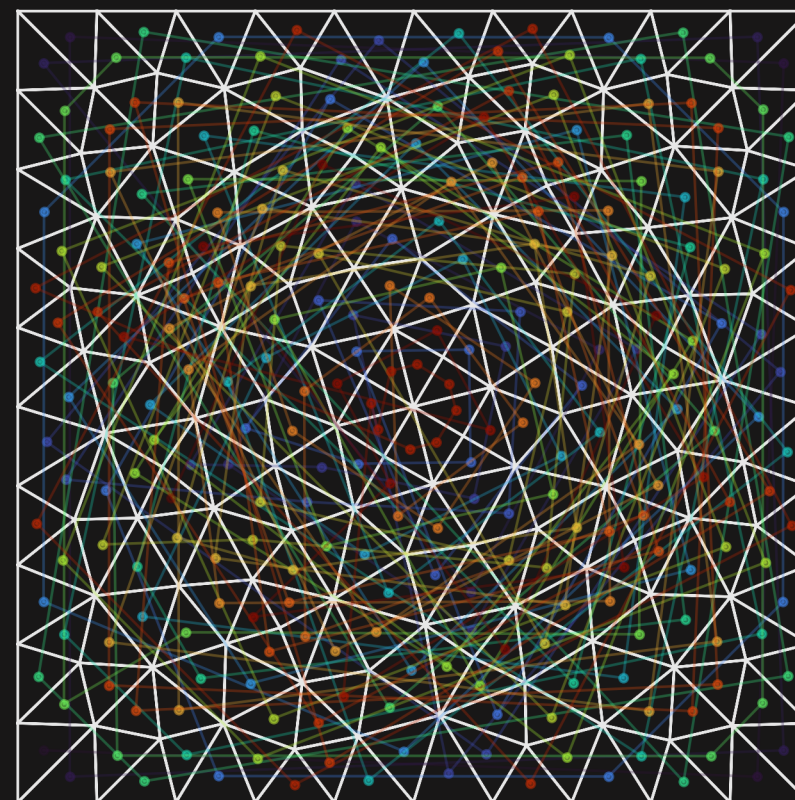
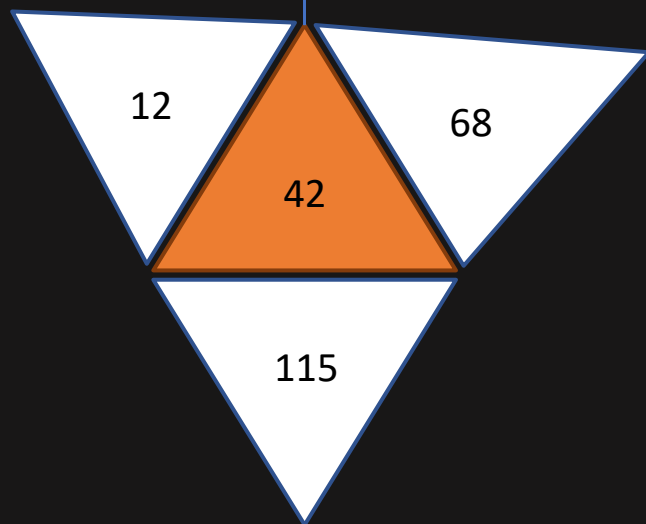
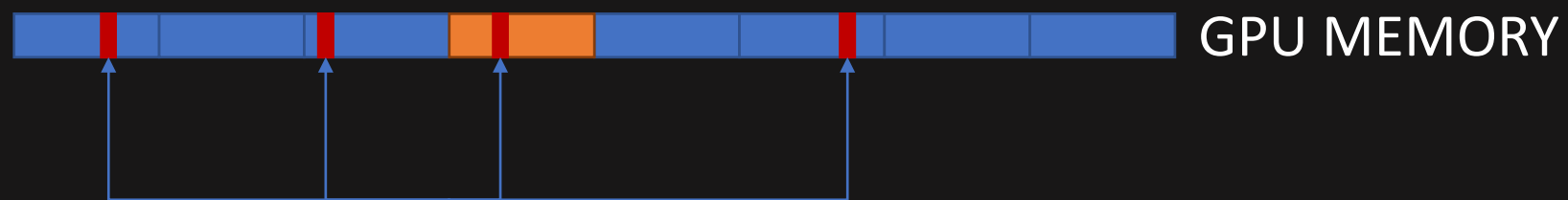
The diagram illustrates the execution flow of a kernel. It is divided into three stages, each represented by a bracket on the right side of the code block:

- 55%**: Loading data (local_u[...] = u[...];)
- 30%**: Heavy processing (represented by a blue box labeled "Heavy processing")
- 15%**: Writing back the result (fu[...] = local_fu;)

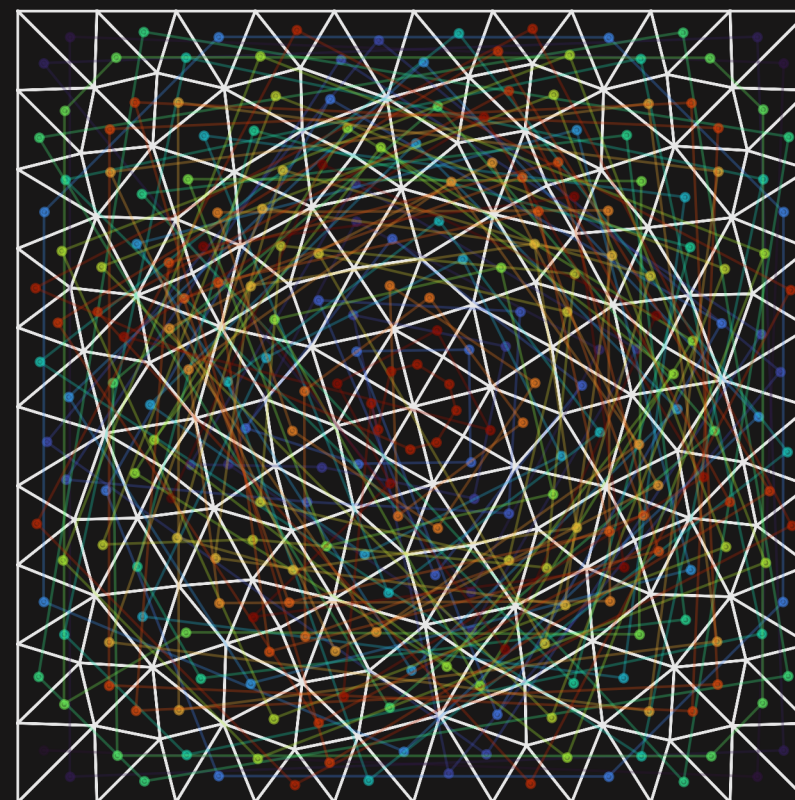
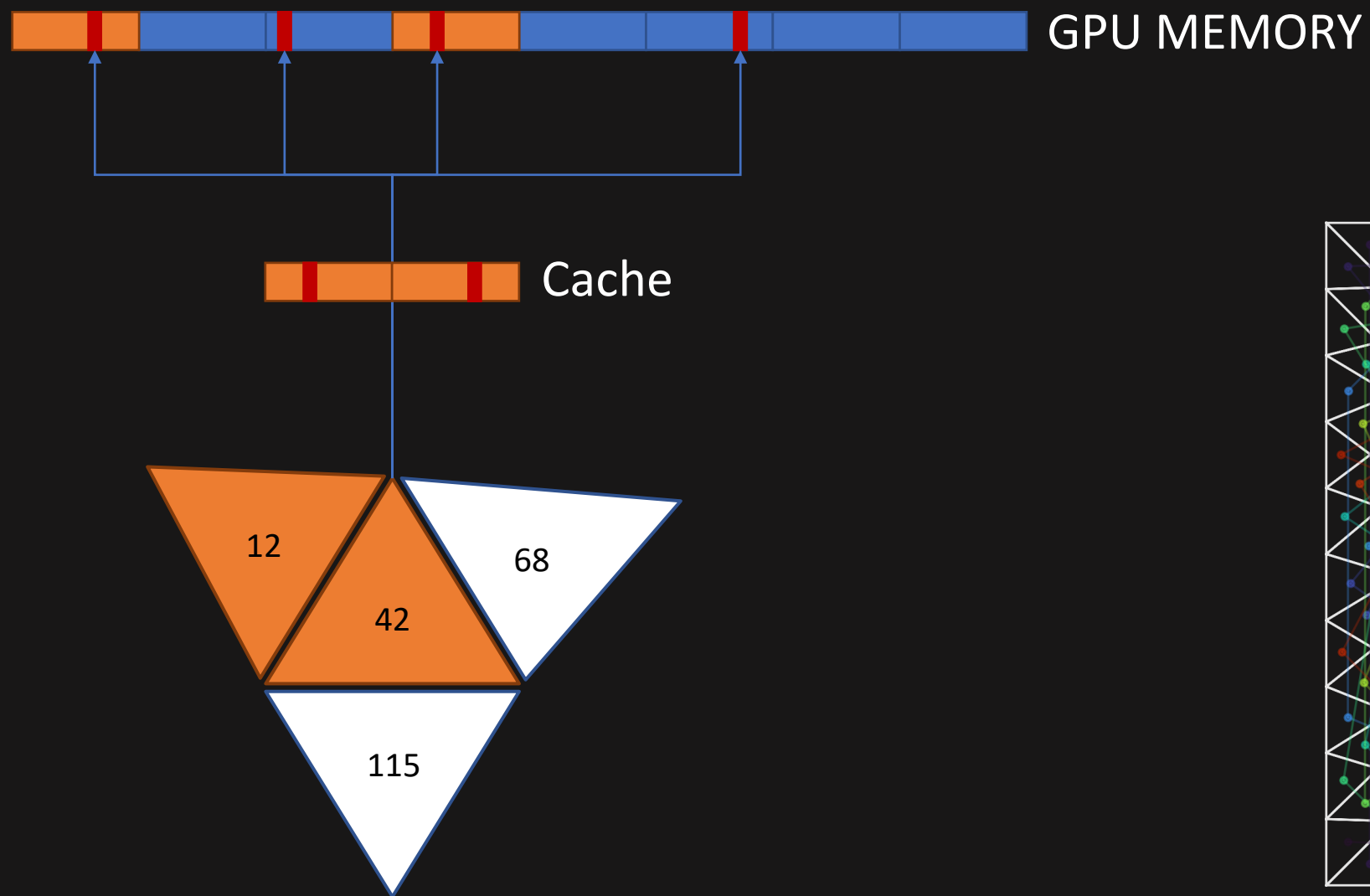
2 – Importance of memory locality



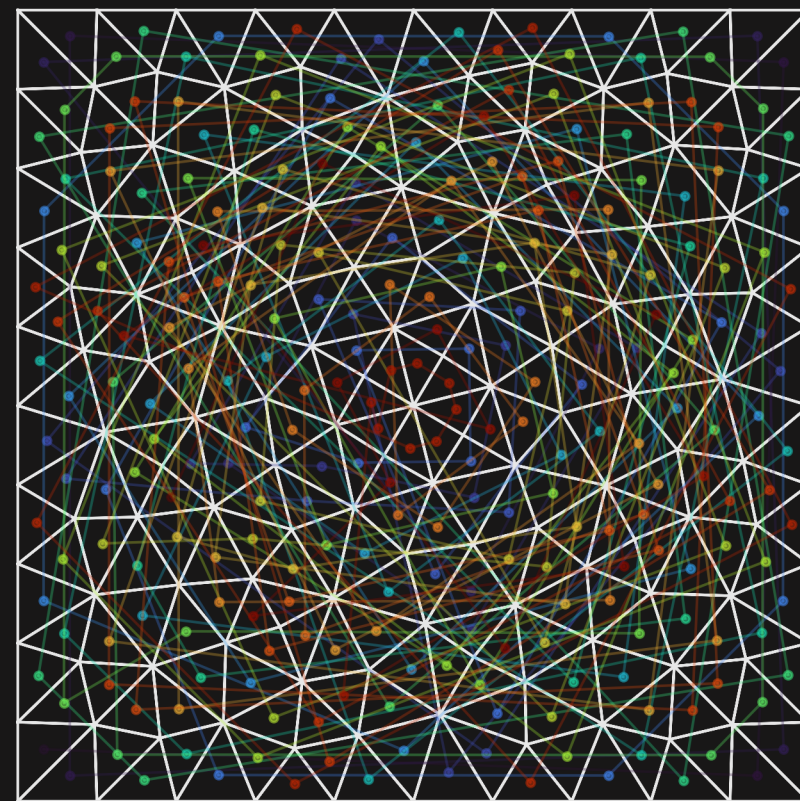
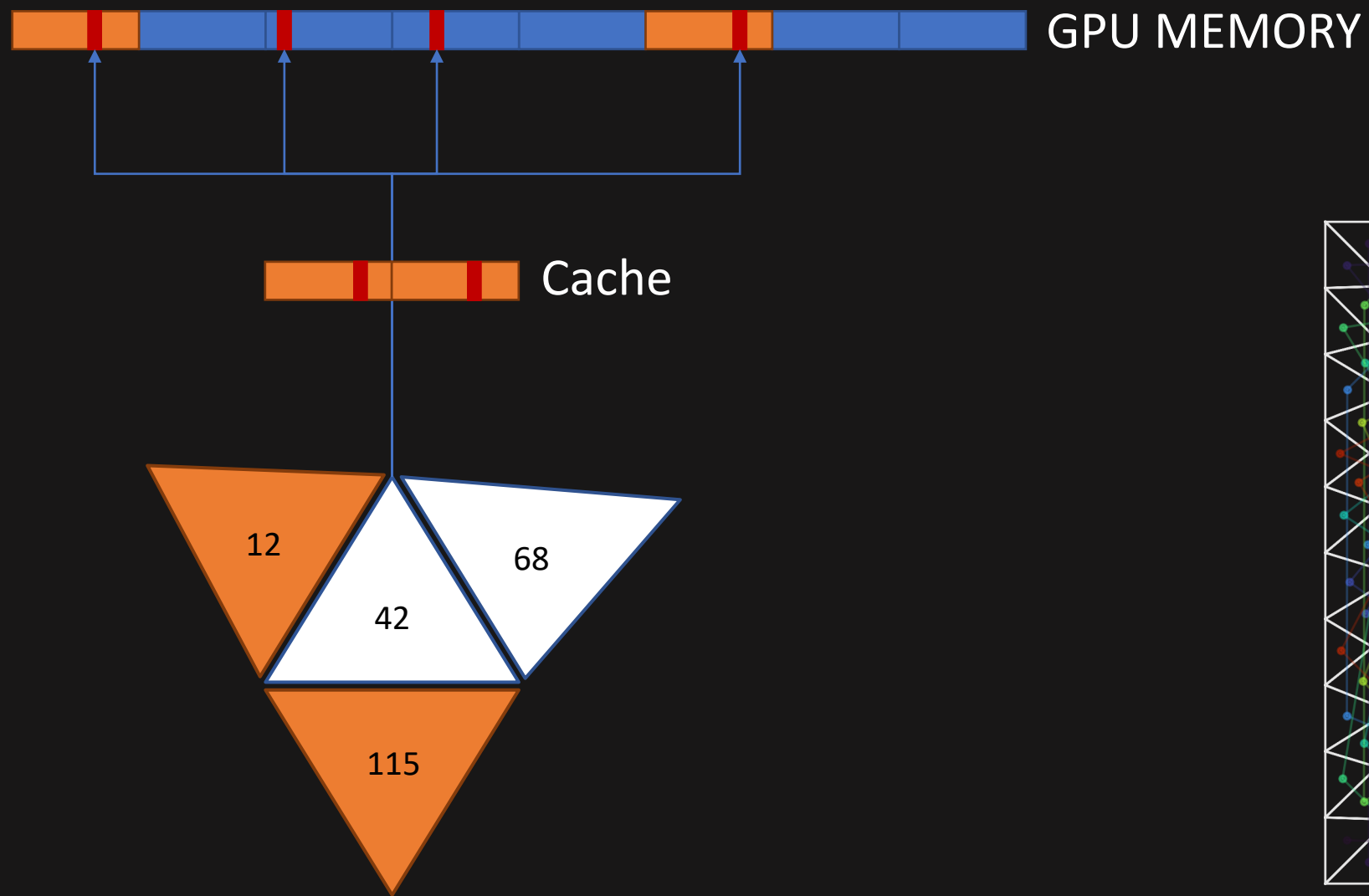
2 – Importance of memory locality



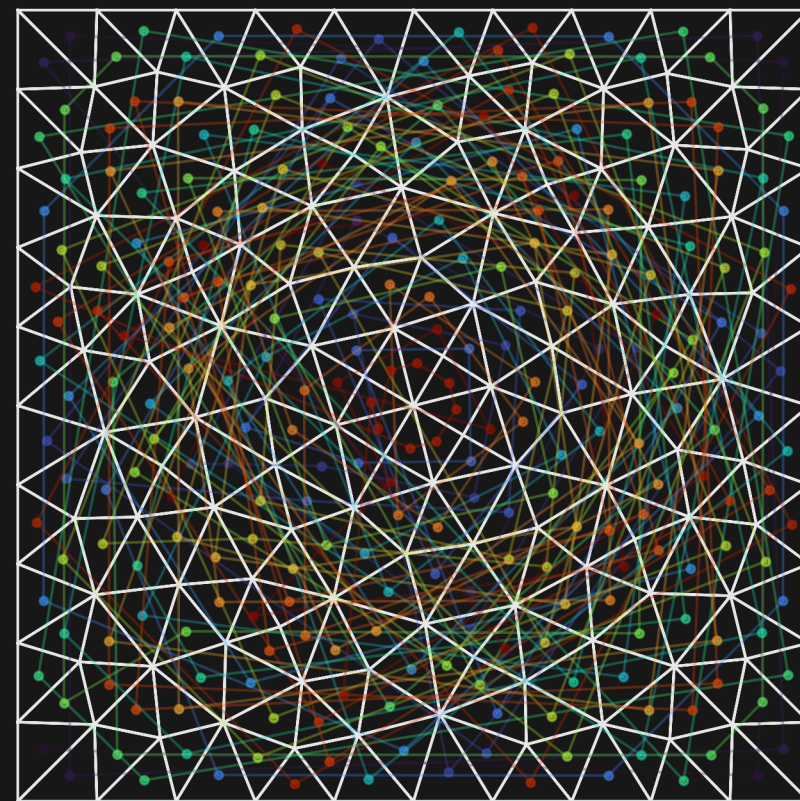
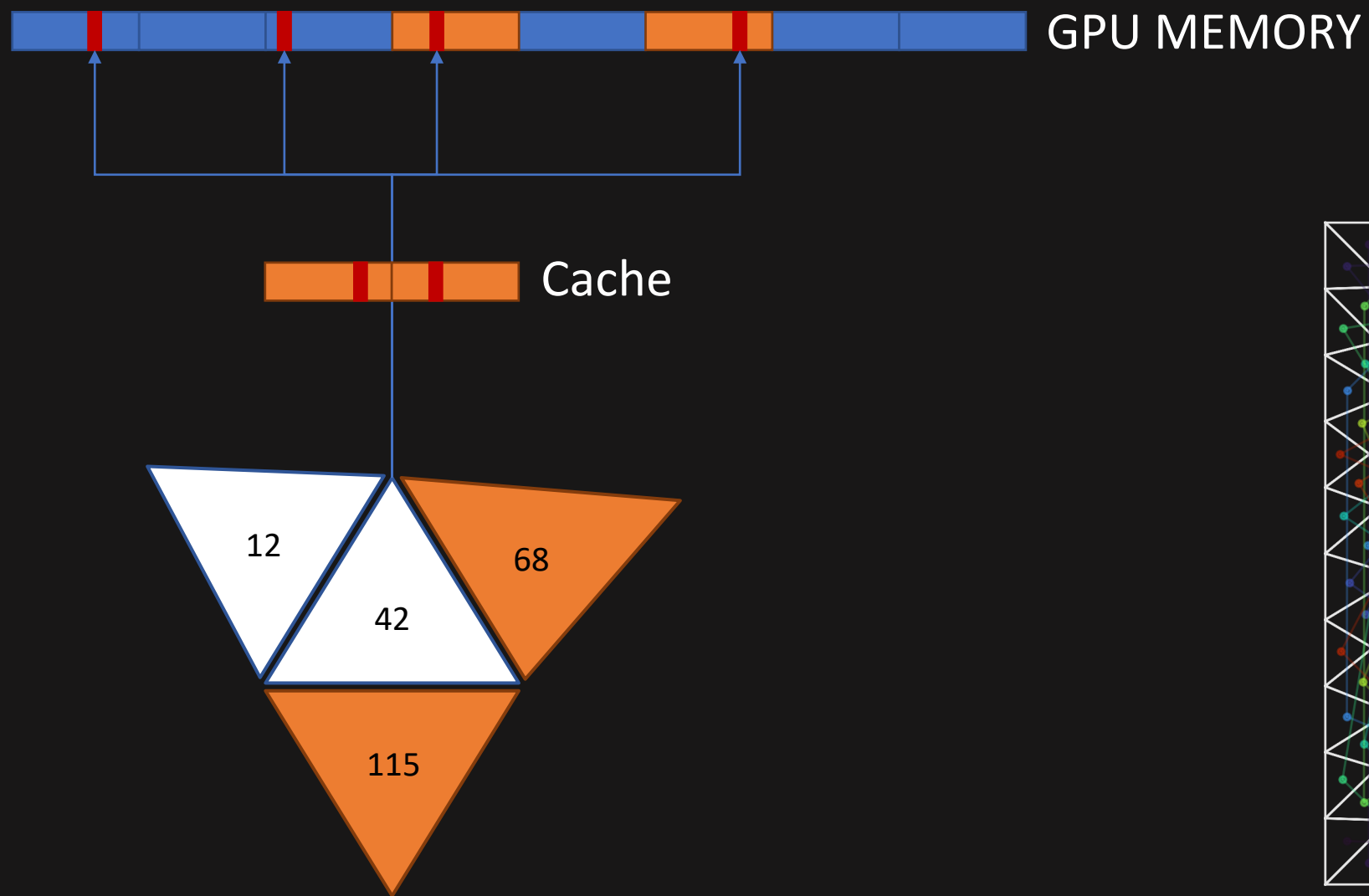
2 – Importance of memory locality



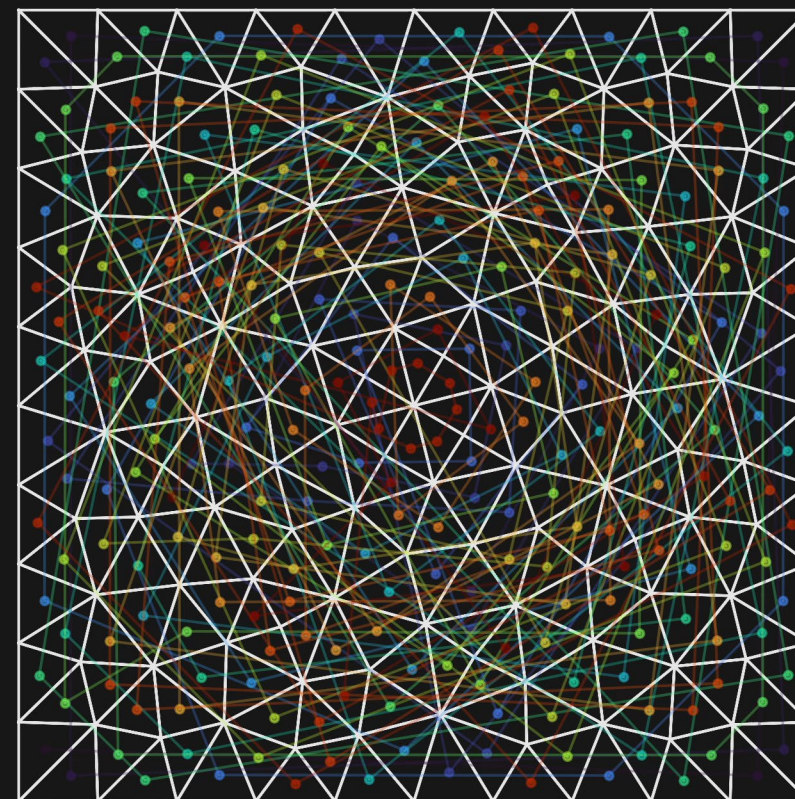
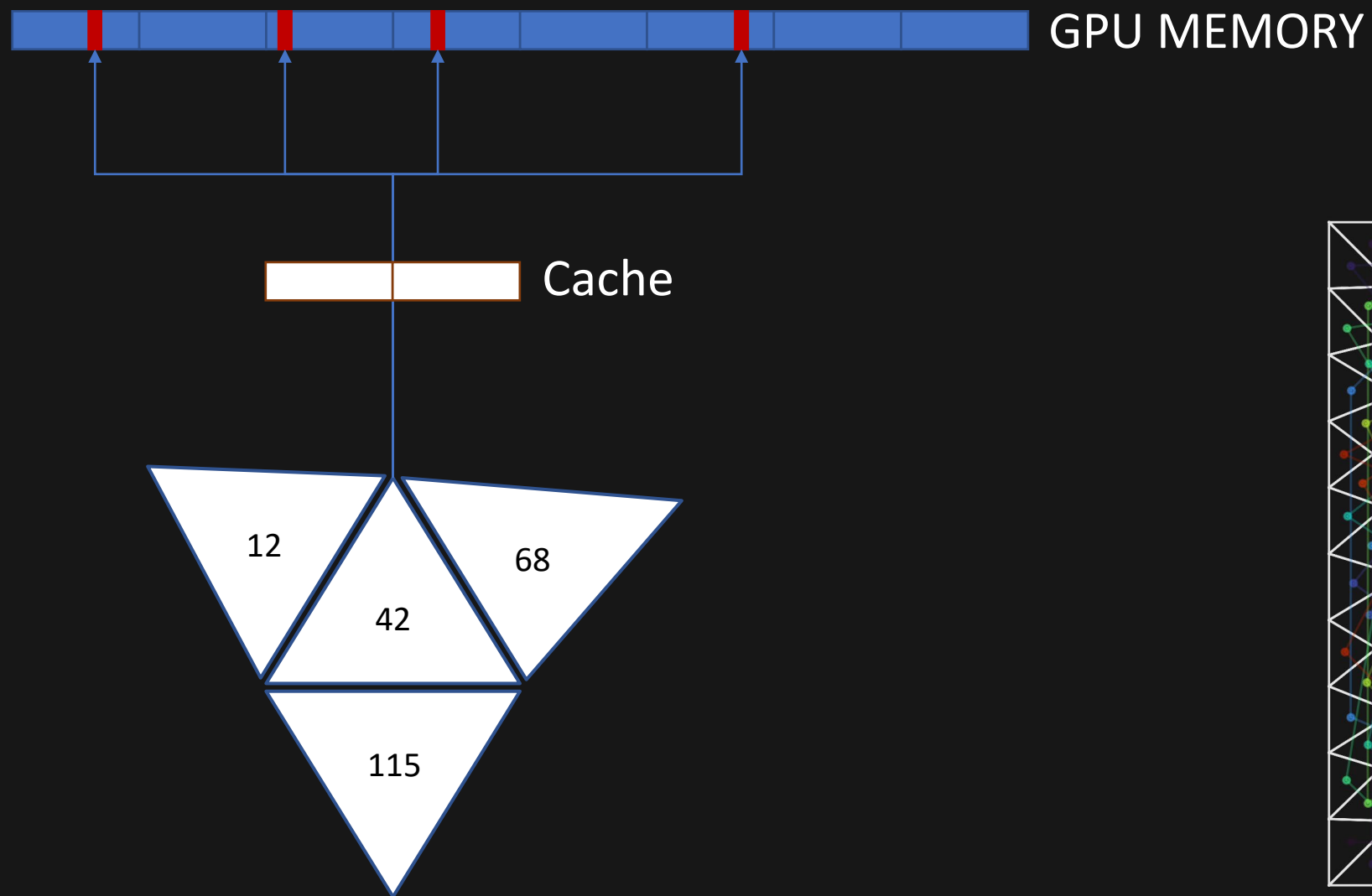
2 – Importance of memory locality



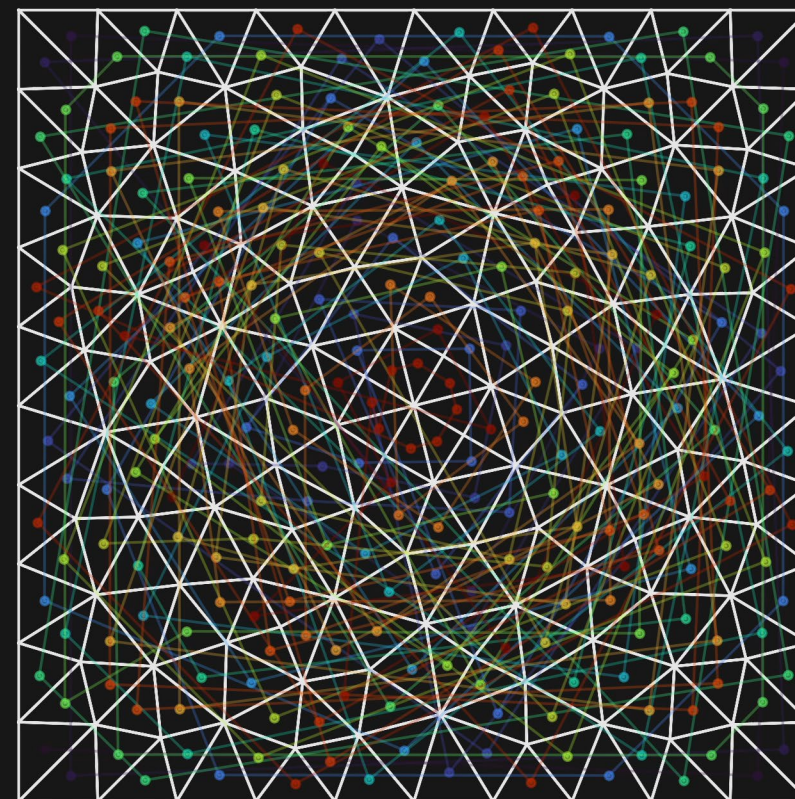
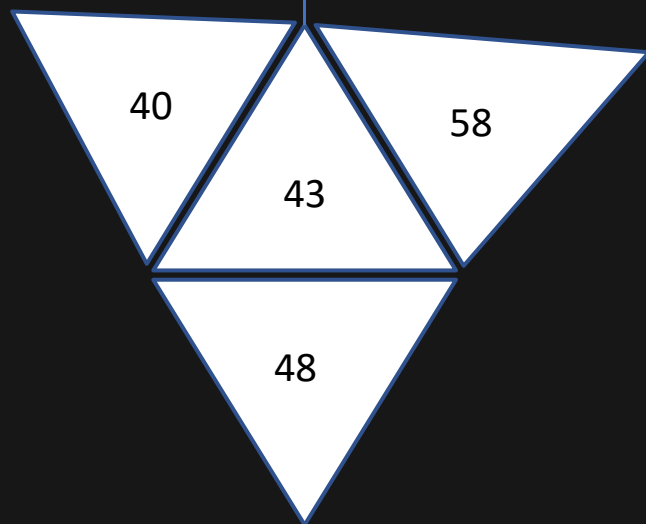
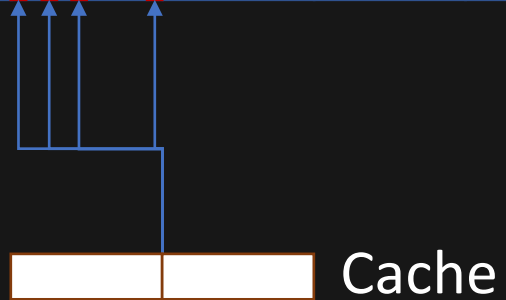
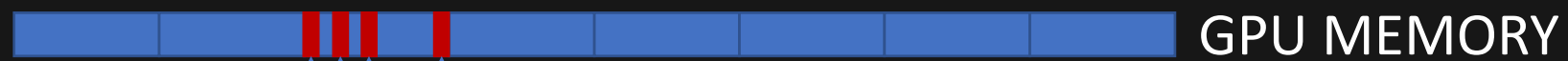
2 – Importance of memory locality



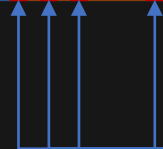
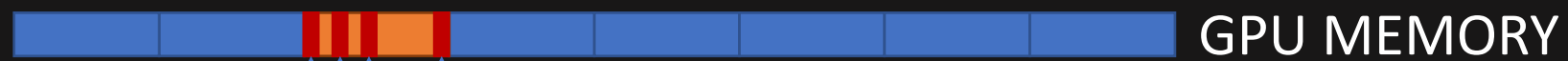
2 – Importance of memory locality



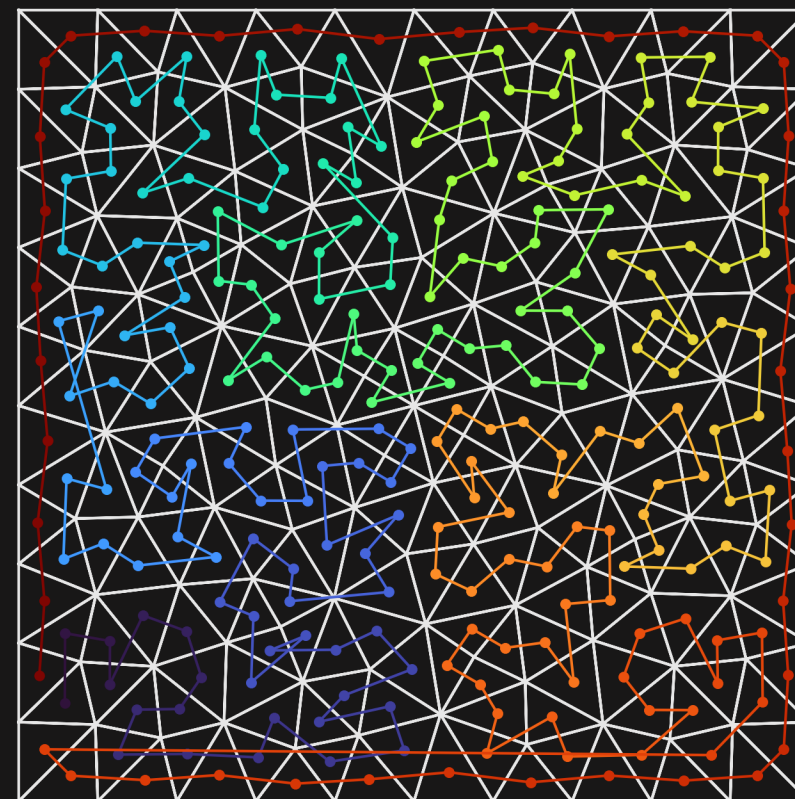
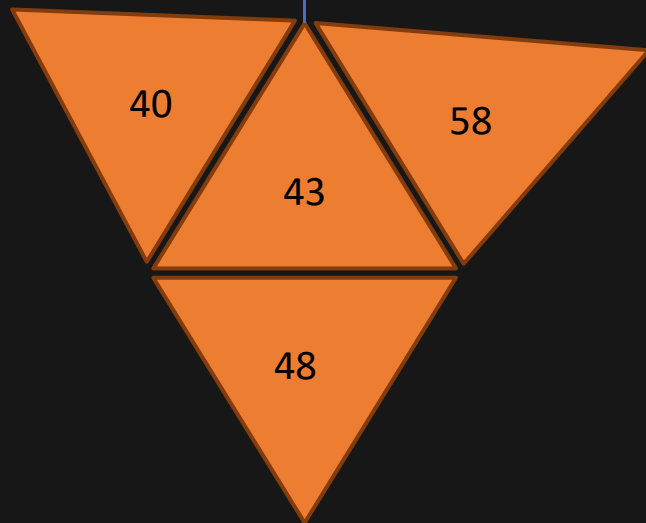
2 – Importance of memory locality



2 – Importance of memory locality



Cache



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.

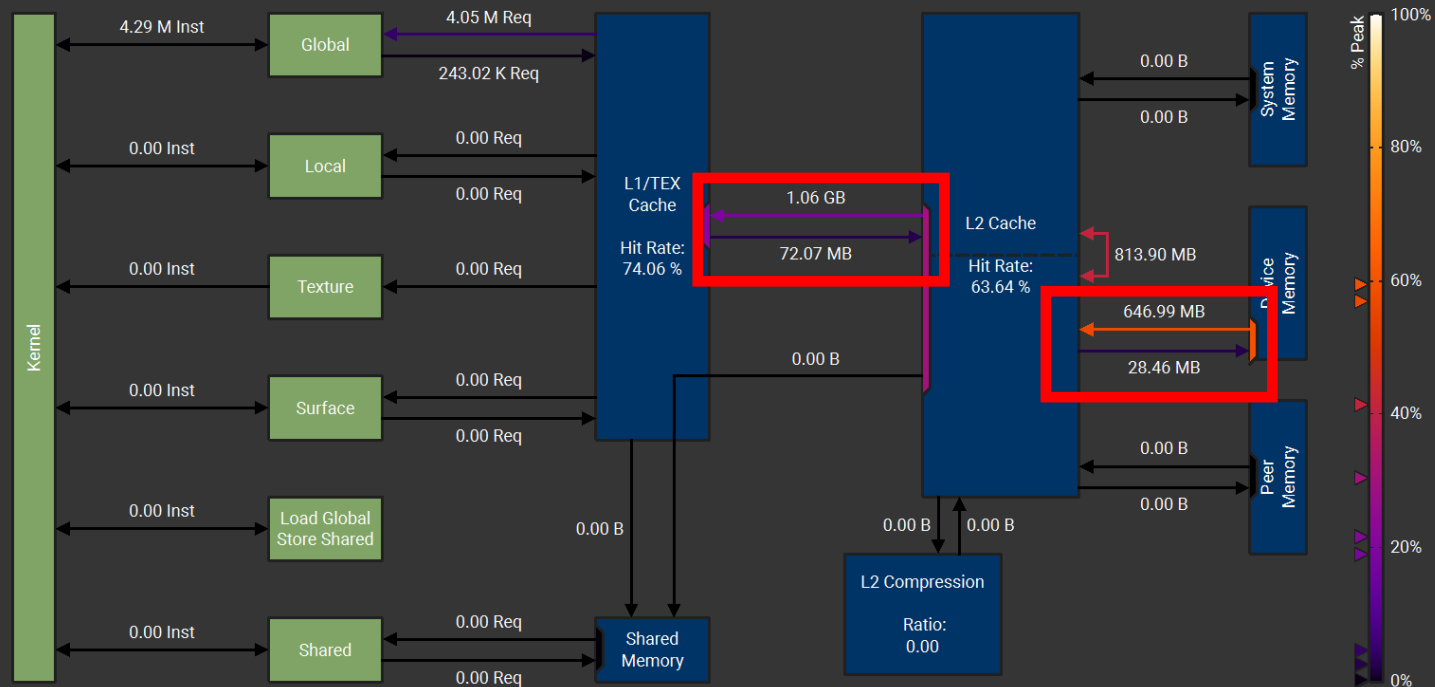
DRAM Excessive Read Sectors The memory access pattern for loads from device memory causes 21,733,744 sectors to be read from DRAM, which is 1.8x of the 12,160,697 sectors causing a miss in the L2 cache. The DRAM fetch granularity for read misses in L2 is 64 bytes, i.e. the lower or upper half of an L2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

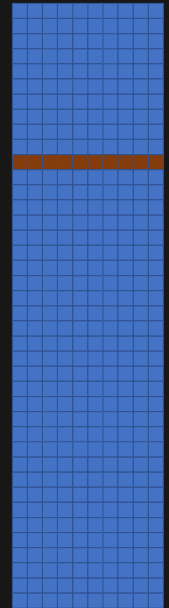
Memory Chart

Show As: Transfer Size

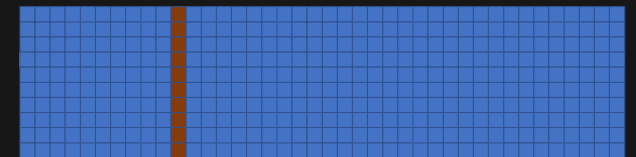


3 – So... let's optimize for locality, right ?

```
__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

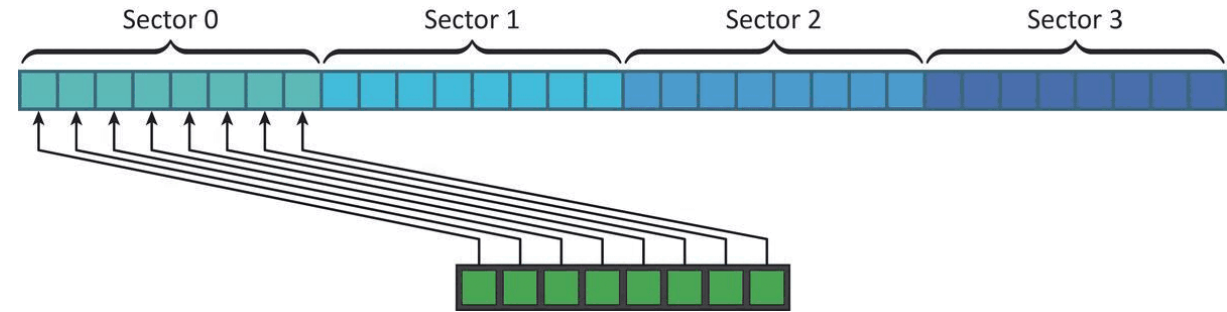
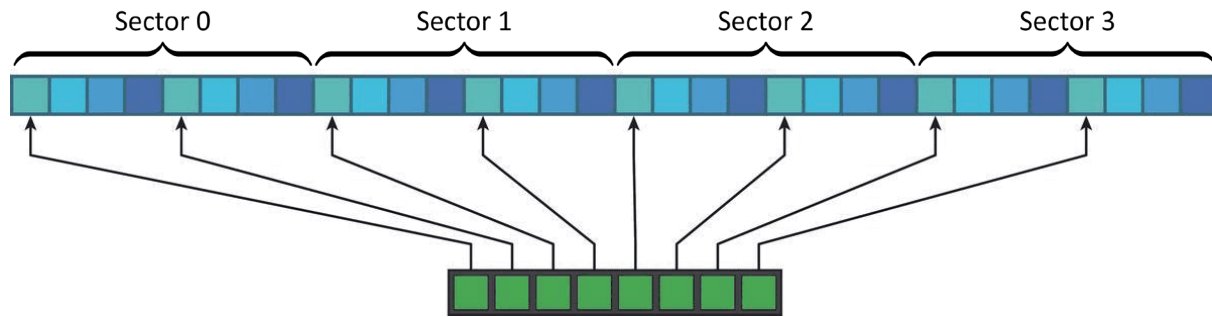
```
__global__ void mysum(const float* a, const float* b, float* c, int nf, int n){  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    if (tid < n){  
        for(int field = 0; field < nf; field++){  
            // maximal locality  
            c[tid * nf + field] = a[tid * nf + field] +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

634 μ 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 μ s

3 – No, we need to consider coalescence

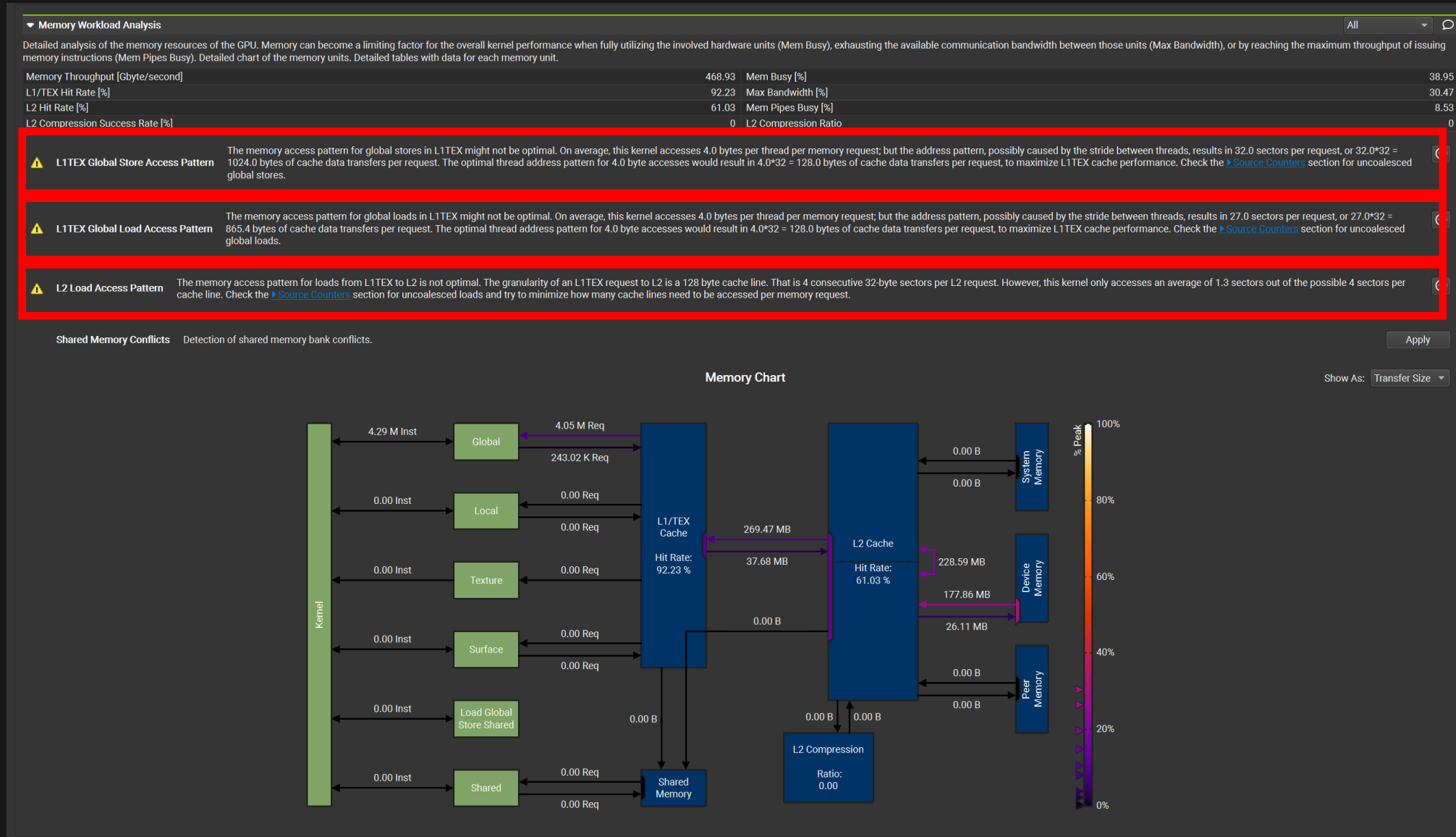


```
__global__ void mysum(const float* a, const float* b, float* c, int nf, int n){  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    if (tid < n){  
        for(int field = 0; field < nf; field++){  
            // maximal locality  
            c[tid * nf + field] = a[tid * nf + field] +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



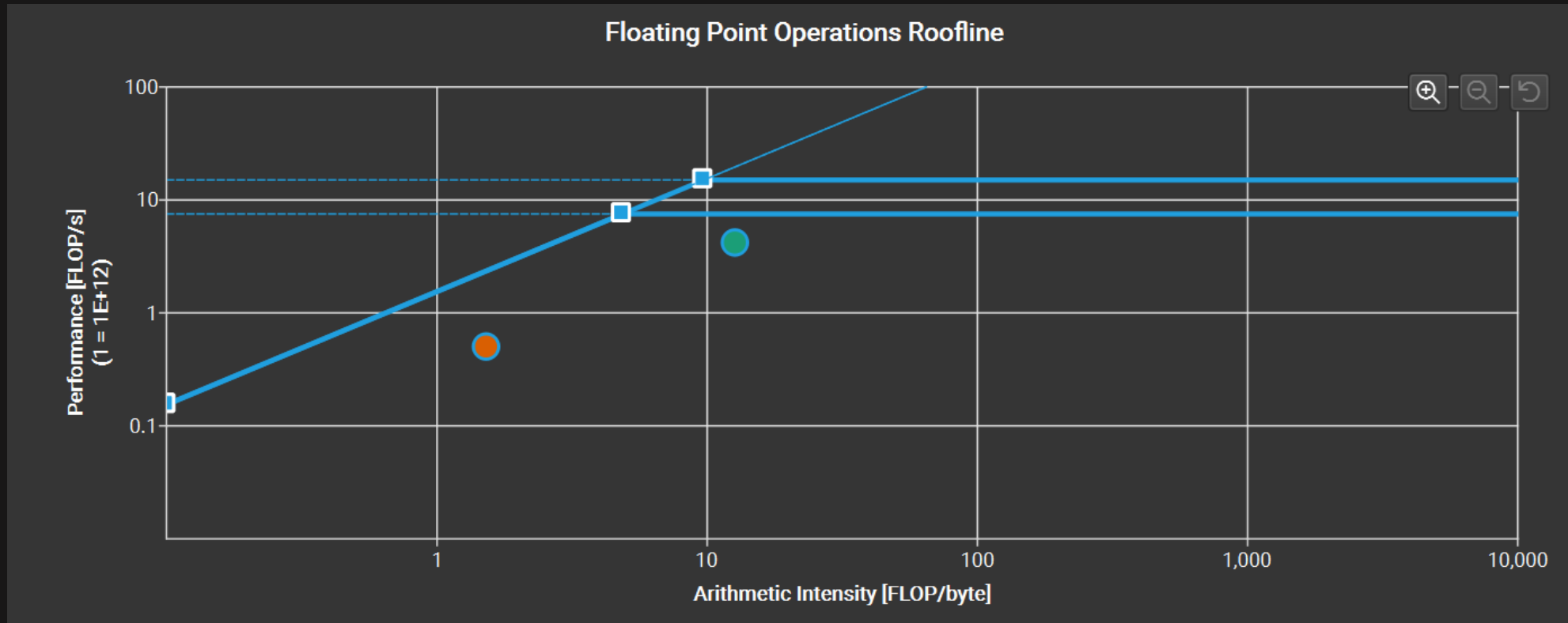
4 – Beware of doubles

```
__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. At the top, the 'Source' dropdown is highlighted with a red box. Below it, the assembly view shows instructions with their addresses and sources. A red box highlights the instruction `DFMA R4, R4, 0.5, R12` at address 406. Another red box highlights the corresponding source code line `scalar un = (unl+unr)/2.0;` in the source view. The interface also shows various performance metrics like Time, Cycles, and GPU usage.

#	Address	Source	Live Registers	arp	Stall	Sampl (All Cycl
390	00007fe9 62fbb550	MOV R4, 0x1870	42			0.07%
391	00007fe9 62fbb560	CALL.REL.NOINC 0x7fe962fbe7e0	166			0.07%
392	00007fe9 62fbb570	MOV R17, R43	49			0.24%
393	00007fe9 62fbb580	BSYNC B0	48			0.08%
394	00007fe9 62fbb590	FMUL R8, R8, R17	48			0.30%
395	00007fe9 62fbb5a0	F2F.F64.F32 R12, R22	50			0.03%
396	00007fe9 62fbb5b0	FADD R11, R18, R3	50			0.01%
397	00007fe9 62fbb5c0	FFMA R8, R7, R10, R8	50			0.05%
398	00007fe9 62fbb5d0	FADD R18, R18, -R3	49			0.05%
399	00007fe9 62fbb5e0	FADD R10, R17, R10	48			0.17%
400	00007fe9 62fbb5f0	F2F.F64.F32 R4, R8	49			0.05%
401	00007fe9 62fbb600	FMUL R9, R9, R18	48			0.03%
402	00007fe9 62fbb610	FMUL R10, R10, 0.5	48			0.06%
403	00007fe9 62fbb620	F2F.F64.F32 R20, R23	50			0.16%
404	00007fe9 62fbb630	FSETP.GT.AND P0, PT, R10, RZ, PT	49			0.01%
405	00007fe9 62fbb640	F2F.F64.F32 R14, R11	48			0.12%
406	00007fe9 62fbb650	DFMA R4, R4, 0.5, R12	48			0.03%
407	00007fe9 62fbb660	F2F.F64.F32 R14, R11	48			0.09%
408	00007fe9 62fbb670	FFMA R19, R10, R19, RZ	47			0.07%
409	00007fe9 62fbb680	F2F.F64.F32 R12, R18	48			0.16%
410	00007fe9 62fbb690	F2F.F32.F64 R4, R4	47			0.15%
411	00007fe9 62fbb6a0	DFMA R14, R14, 0.5, R20	46			0.03%
412	00007fe9 62fbb6b0	DMUL R14, R94, R14	44			0.14%
413	00007fe9 62fbb6c0	F2F.F64.F32 R6, R6	45			0.07%
414	00007fe9 62fbb6d0	DMUL R12, R14, R12	45			0.13%
415	00007fe9 62fbb6e0	F2F.F64.F32 R32, R4	45			0.17%
416	00007fe9 62fbb6f0	F2F.F64.F32 R8, R9	45			0.18%
417	00007fe9 62fbb700	DFMA R12, R12, -0.5, R32	45			0.16%
418	00007fe9 62fbb710	DFMA R6, R8, 0.5, R6	43			0.21%
419	00007fe9 62fbb720	F2F.F32.F64 R12, R12	41			0.14%

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

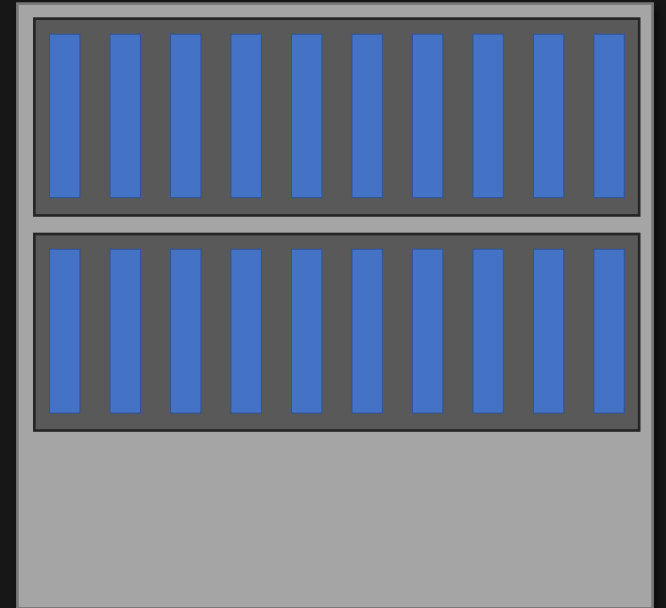
```
__device__ float inv2x2(const float mat[2][2], float
inv[2][2])
{
    float det = det2x2(mat);
    if(det){
        float ud = 1.0 / det;
        inv[0][0] = mat[1][1] * ud;
        inv[1][0] = -mat[1][0] * ud;
        inv[0][1] = -mat[0][1] * ud;
        inv[1][1] = mat[0][0] * ud;
    }
    else{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

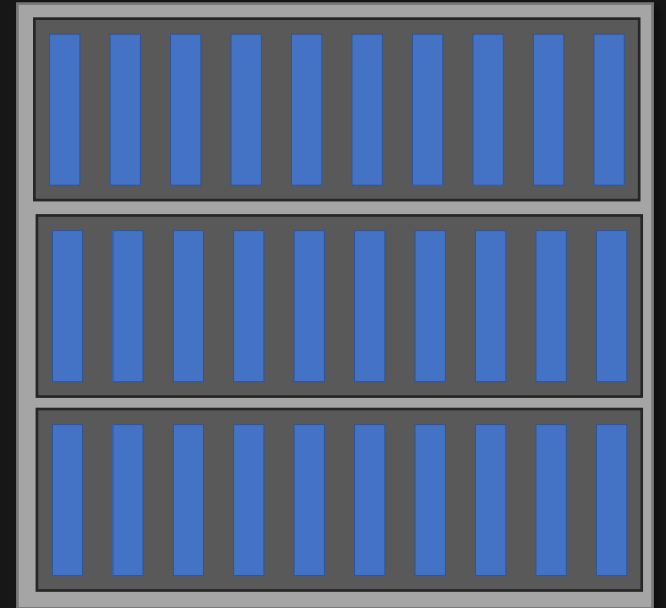

5 – Watch the occupancy!

```
__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 »

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	31.25	Block Limit Registers [block]	5
Theoretical Active Warps per SM [warp]	20	Block Limit Shared Mem [block]	16
Achieved Occupancy [%]	29.69	Block Limit Warps [block]	16
Achieved Active Warps Per SM [warp]	19.00	Block Limit SM [block]	32

Occupancy Limiters This kernel's theoretical occupancy (31.2%) is limited by the number of required registers See the [CUDA Best Practices Guide](#) for more details on optimizing occupancy.

Achieved Occupancy Analysis of the Achieved Occupancy Apply

Theoretical Occupancy Analysis of Theoretical Occupancy and its Limiters Apply

Impact of Varying Register Count Per Thread

Registers Per Thread	Warp Occupancy
8	72
16	72
24	72
32	72
40	48
48	45
56	42
64	40
72	38
80	36
96	24
104	22
112	21
120	20
128	18
136	17
144	16
152	15
160	14
168	12
176	11
184	10
192	9
200	8
208	7
216	6
224	5
232	4
240	3
248	2
256	1

Impact of Varying Block Size

Block Size	Warp Occupancy
32	12
64	20
96	18
128	24
160	20
192	18
224	12
256	15
288	18
320	20
352	12
384	12
416	13
448	14
480	15
512	16
544	17
576	18
608	19
640	20
672	0
704	0
736	0
768	0
800	0
832	0
864	0
896	0
928	0
960	0
992	0
1,024	0

Impact of Varying Shared Memory Usage Per Block

Shared Memory Per Block	Warp Occupancy
5,232	20
10,464	10
15,696	5
20,928	5
26,160	5
31,392	5
36,624	5
41,856	5
47,088	5
52,320	5
57,552	5
62,784	5
68,016	5
73,248	5
78,480	5
83,712	5
88,944	5
94,176	5
99,408	5
104,640	5
109,872	5
115,104	5
120,336	5
125,568	5
130,800	5
136,032	5
141,264	5
146,496	5
151,728	5
156,960	5
162,192	5
167,424	0

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 is labeled 'dudt', and the third is labeled 'axpy'. The fourth segment is empty. The segments are separated by thin white vertical lines.

axpy

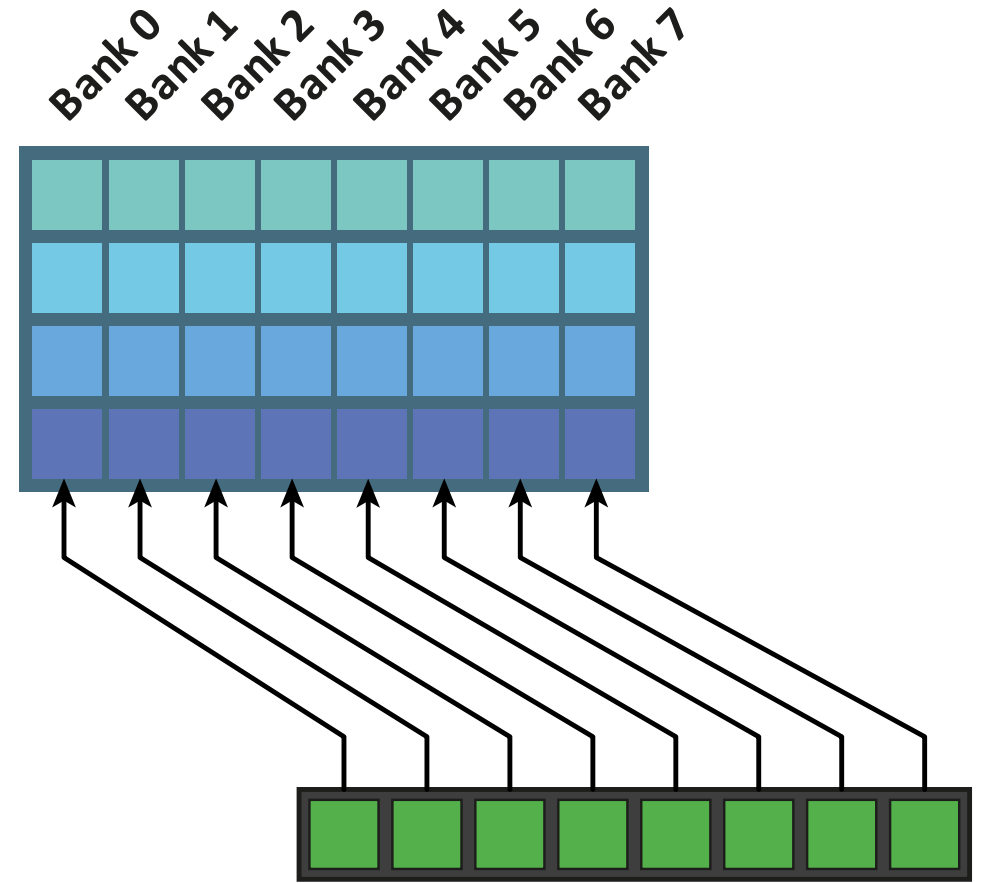
dudt

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
- `--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.