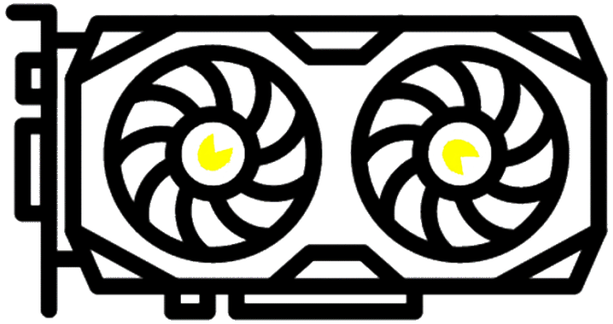
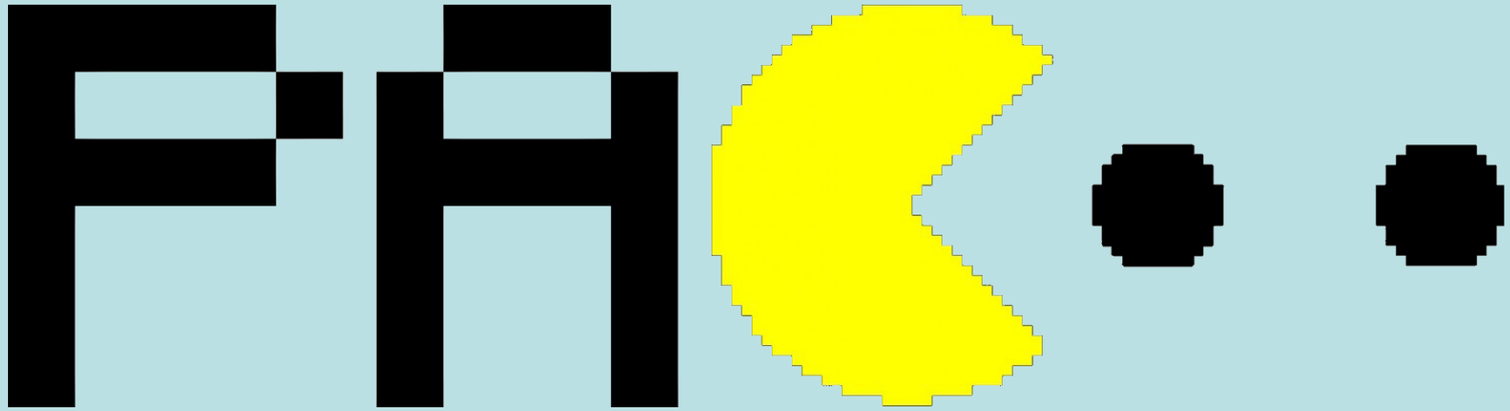
University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

member of
swissuniversities

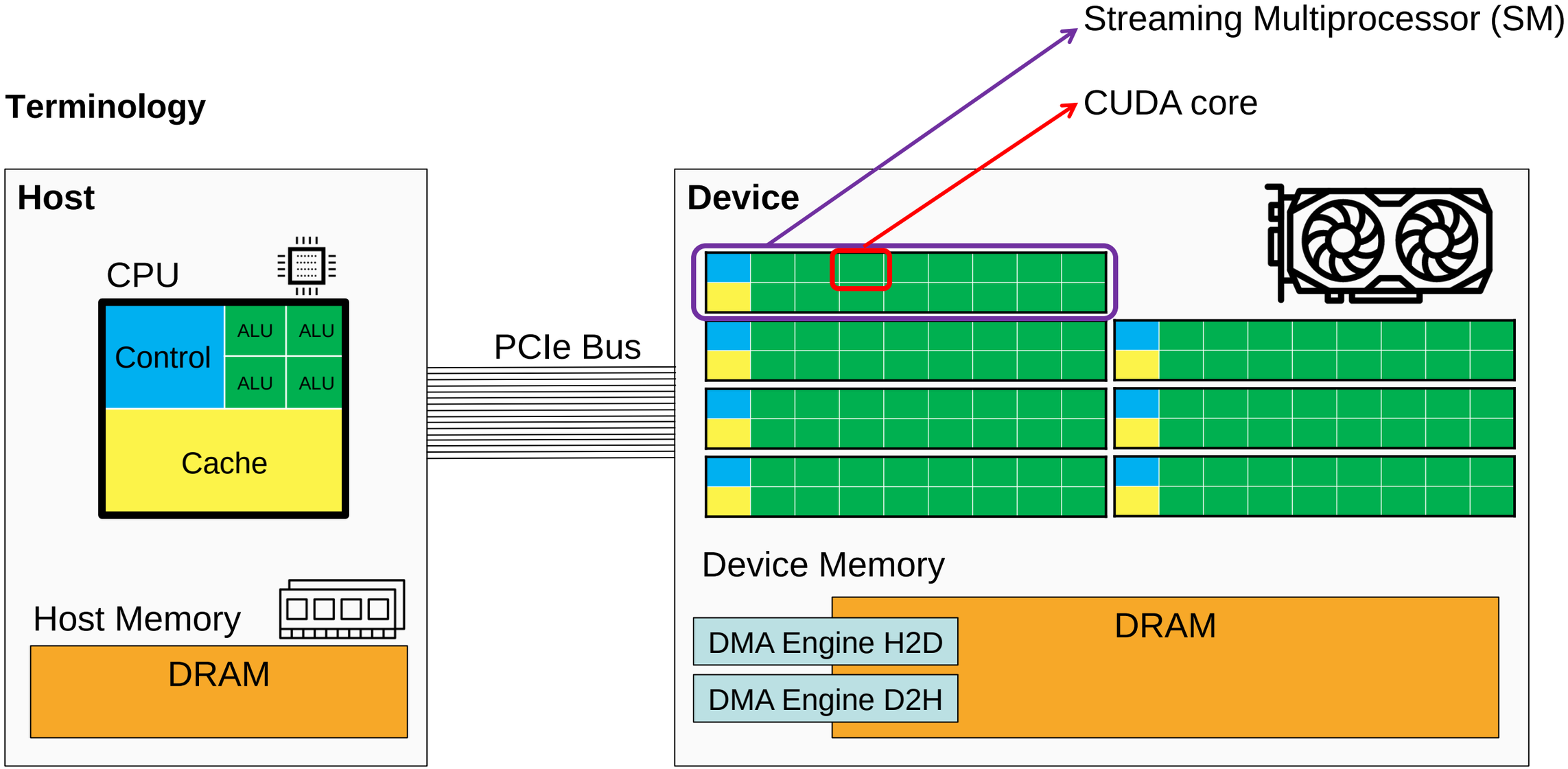


Parallel Computing
GPU Program Flow



Simon Marcin, Filip Schramka

Terminology

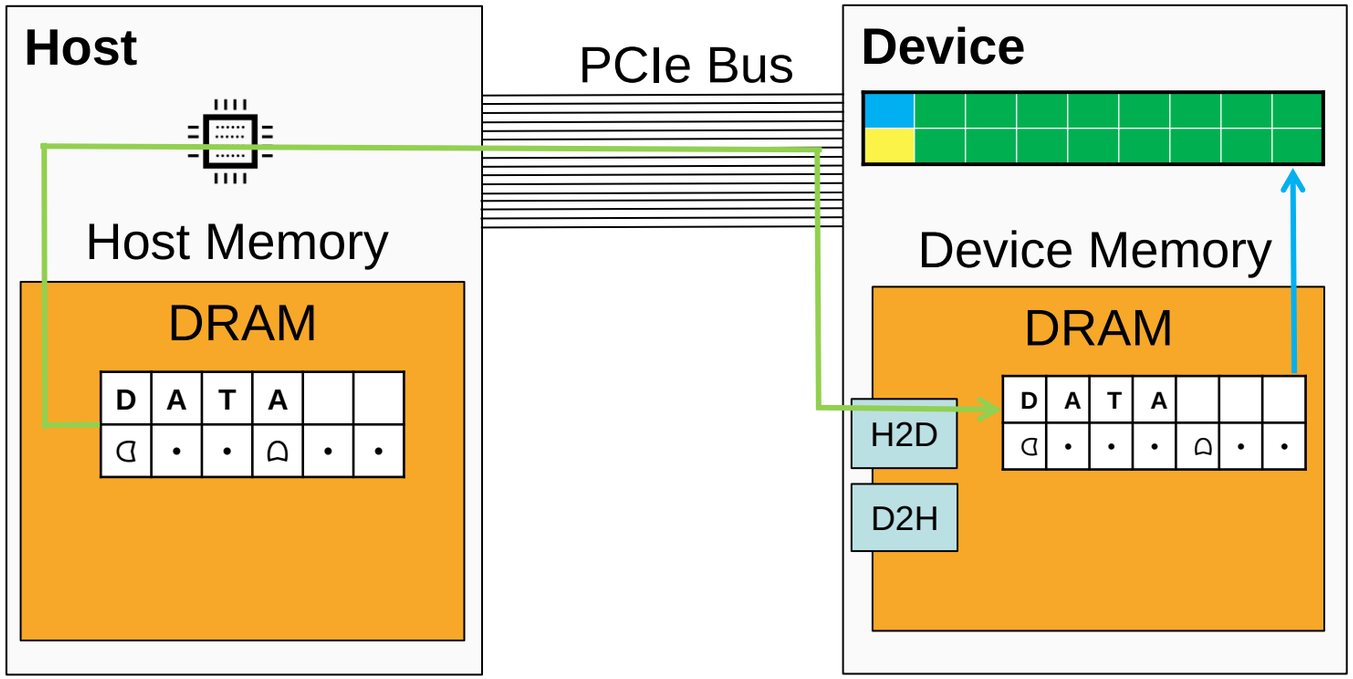


CUDA Processing Flow

1. Load data into Host Memory
   - CPU load

2. Copy data to Device using H2D
   - CPU load
   - H2D engine load

3. Execute kernel
   - GPU load (kernel engine)



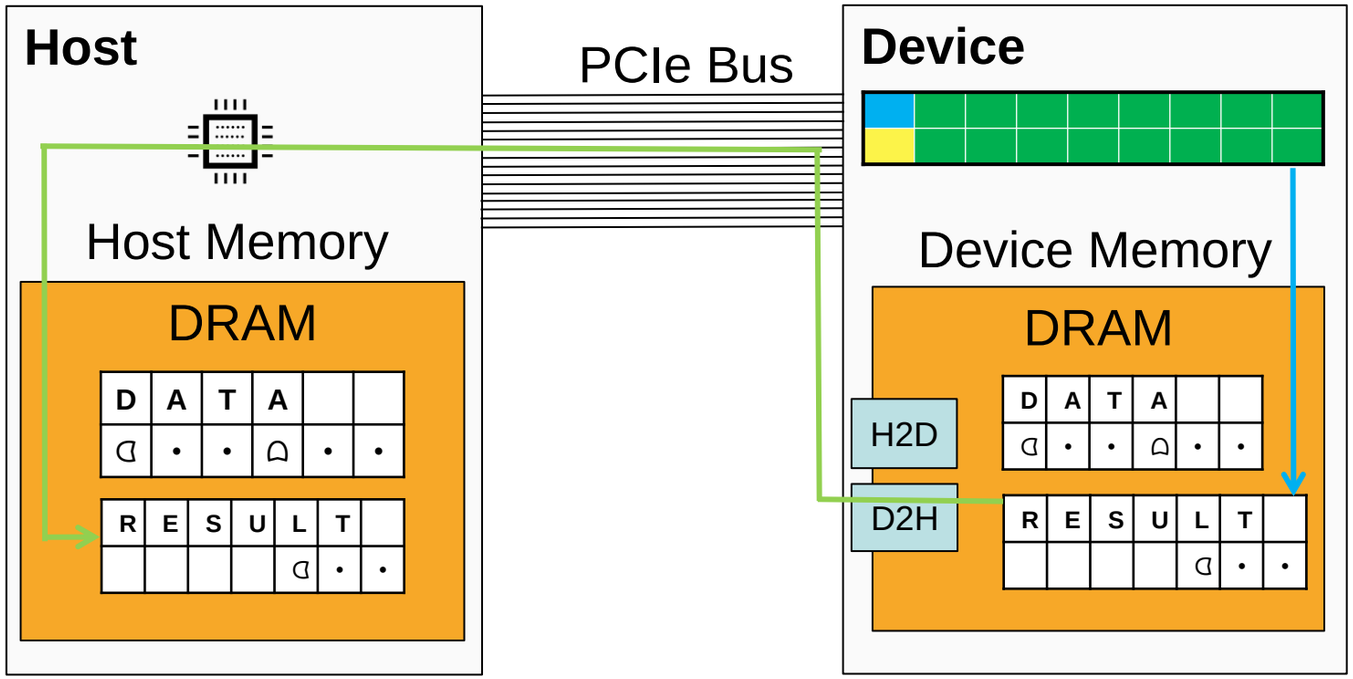
* Allocation step of memory is skipped in this drawing

CUDA Processing Flow

3. Execute kernel
   - Create result data

4. Copy result to Host using D2H
   - CPU load
   - D2H engine load

5. Do something with the result
   - CPU load



* Allocation step of memory is skipped in this drawing

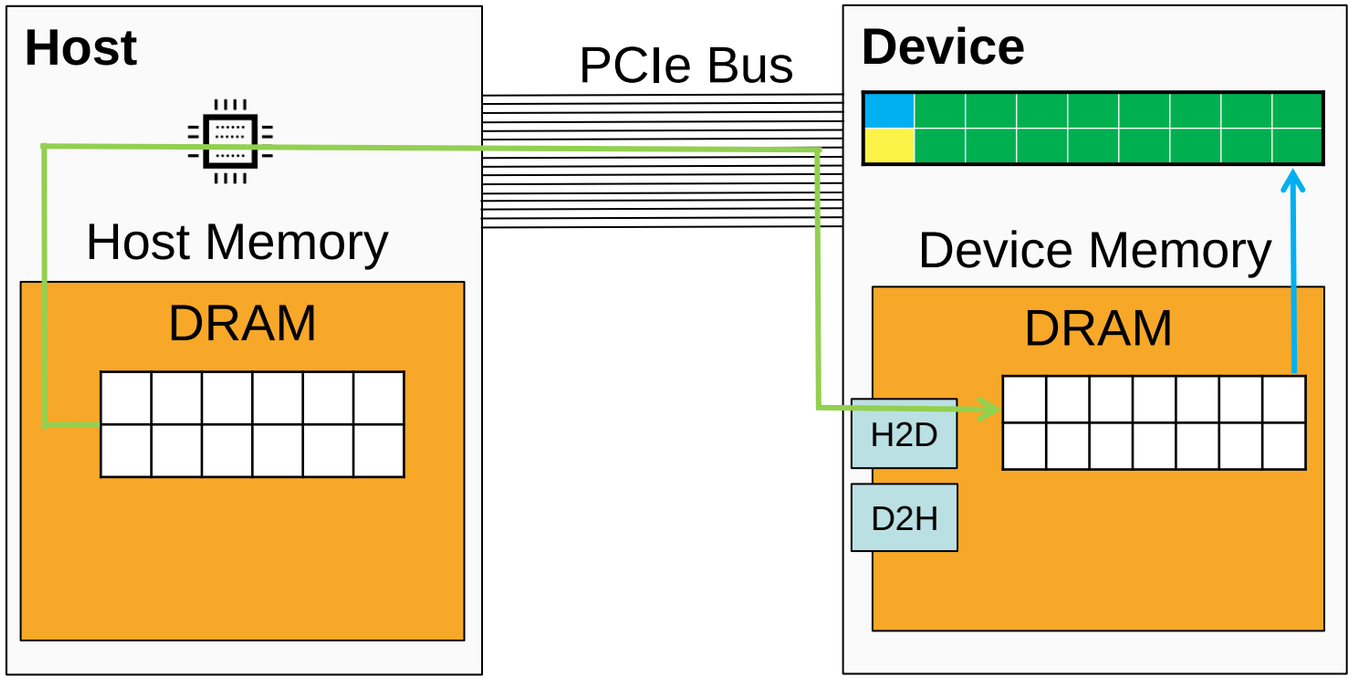
CUDA Processing Flow

```
// Allocate host memory
int N = 1 << 20;
int* hostVectorA = new int[N];

// Allocate device memory
int* deviceVectorA;
cudaMalloc(&deviceVectorA,
           N * sizeof(int));

// Copy data from host to device
cudaMemcpy(deviceVectorA,
           hostVectorA,
           N * sizeof(int),
           cudaMemcpyHostToDevice);

// Run kernel “cudaAdd” on the GPU
cudaAdd <<<1, 1 >>> (deviceVectorA, ..., N);
```

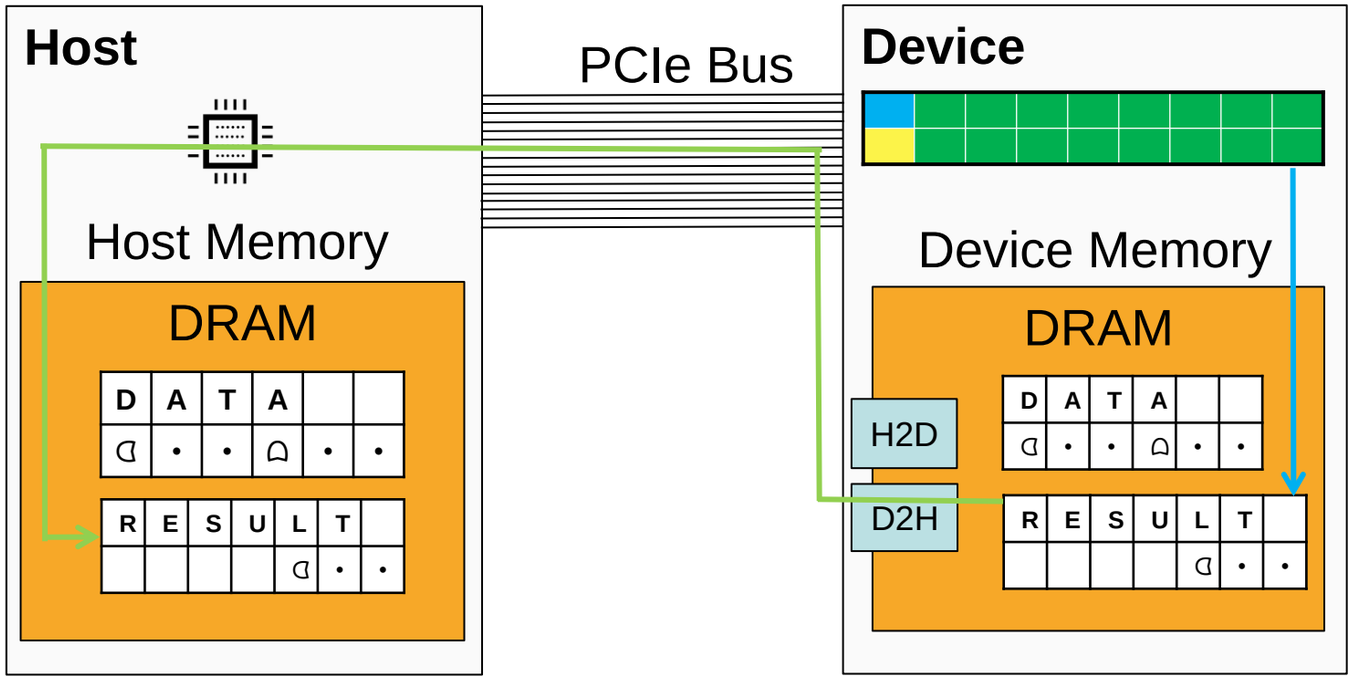


CUDA Processing Flow

```
// Copy data from device to host
cudaMemcpy(hostVectorCGPU,
           deviceVectorC,
           N * sizeof(int),
           cudaMemcpyDeviceToHost);

// Free memory on device
cudaFree(deviceVectorA);

// Free memory on host
delete[] hostVectorA;
```



CUDA Processing Flow

- All CUDA API calls return an error/success code
- We will use a C Macro to check if the CUDA calls worked
- Async CUDA calls (like kernel exec) need to be checked explicitly

```
// Copy data from device to host
gpuErrCheck(cudaMemcpy(...));

// Run kernel “cudaAdd” on the GPU
cudaAdd <<<1, 1 >>> (deviceVectorA, ..., N);
gpuErrCheck(cudaPeekAtLastError());

// Free memory on device
GpuErrCheck(cudaFree(deviceVectorA));
```

Observations

- The CPU is busy while handling data transfers
- There are 2 different copy engines on the GPU, on for each direction
- The compute kernel on the GPU is idle during data transfer
- All CUDA calls on the GPU are pipelined

06_VectorAddNaive

Task:

- Get the code running using the slides above
- Run the code using the CUDA debugger
- Check the timeline of the execution and its calls using Nvidia Nsight Systems

Link:

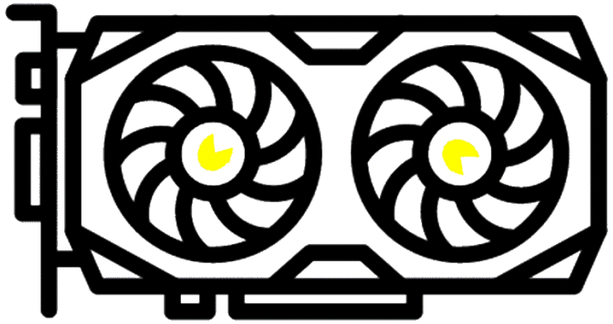
- https://classroom.github.com/a/BTkC-eHe

Goal:

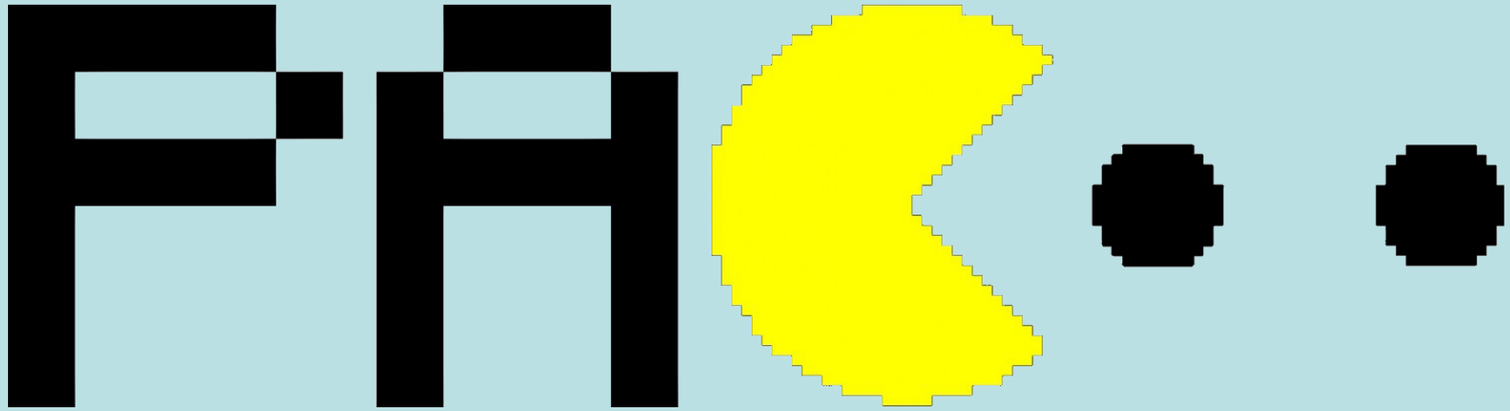
- Get CUDA code running
- Implement a basic CUDA data flow
- Basic knowhow about the Nvidia tools

University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

member of
swissuniversities



Parallel Computing
CUDA execution model

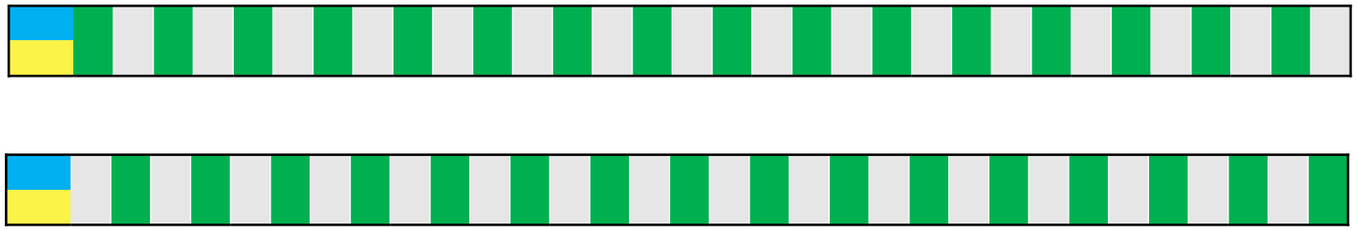


CUDA execution model - SIMD

- The GPU is operating in the Single instruction, multiple data (SIMD) paradigm
- The current GPUs are running the same operation on 32 threads (this is called a warp)
  - E.g: int32 add operation

- If you branch, you lose performance:

```
if (threadId.x % 2 == 0) {
    do_something_A();
} else {
    do_something_B();
}
```



CUDA execution model - Threads

- In order to parallelize the computation, we need lots of threads.
- the number of threads is defined in the kernel execution configuration <<< >>>:

```
kernel_launch<<<blocks, threads>>>(arguments)
```

- There will be multiple SIMD calls for each 32 coalesced threads to work through all the threads

```
kernel_launch<<<blocks, 96>>>(arguments)
// Do one operation for each thread:
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95

CUDA execution model - Threads

- The threads argument can also be 2D or 3D:

```
kernel_launch<<<blocks, dim3(16, 16)>>>(arguments)
kernel_launch<<<blocks, dim3(16, 16, 4)>>>(arguments)
```

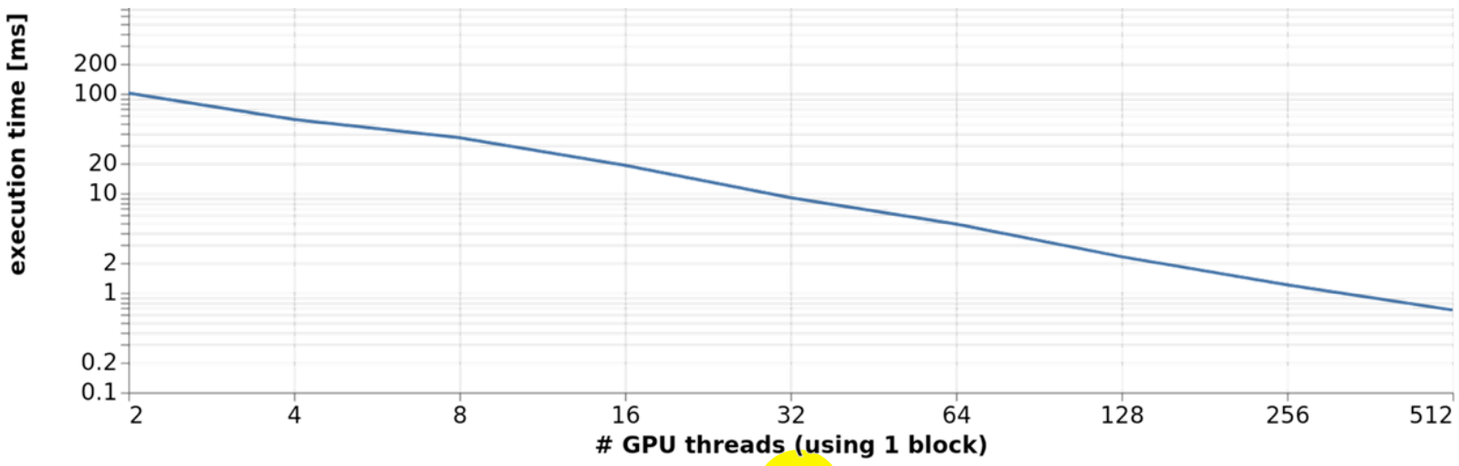
- To identify the thread in the kernel code, use the CUDA provided special object `threadIdx`

```
threadIdx.x
threadIdx.y
threadIdx.z
```

- The number of threads is limited because the number of registers that can be allocated is limited
- The number of threads should be a multiple of 32 as the warp size is 32.
- Max dimension size for `(x, y, z)` is `(1024, 1024, 64)` where `x * y * z ≤ 1024`
  See https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications for the exact specs of the different architectures

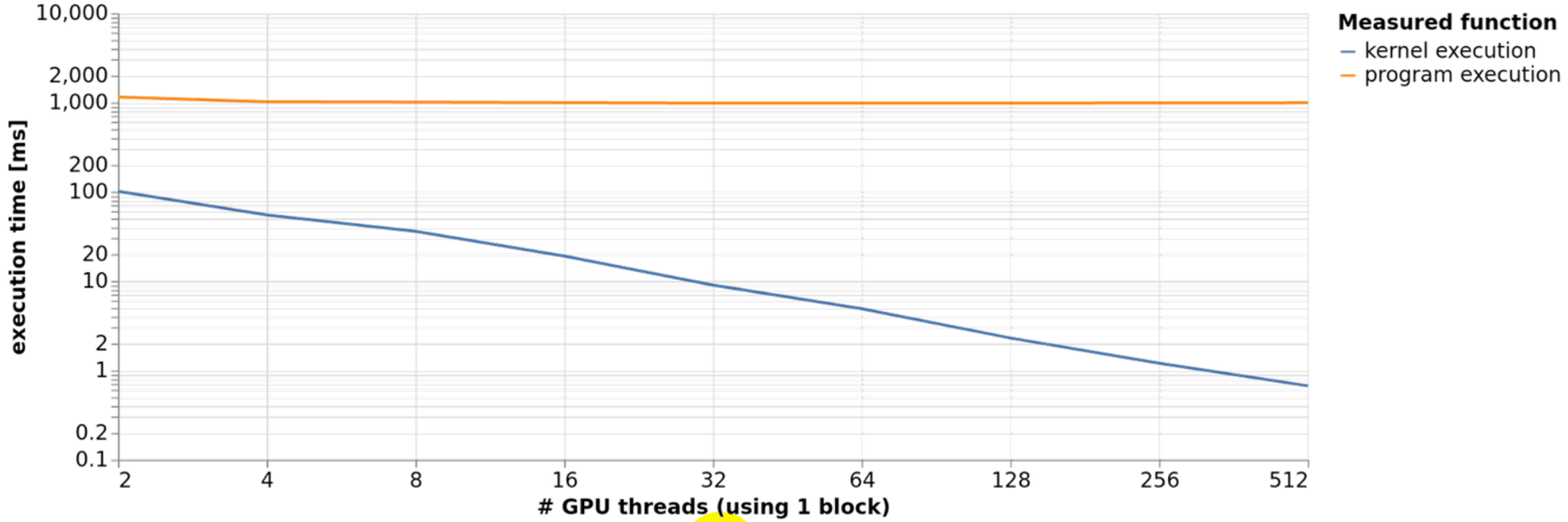
CUDA execution model - Threads

- Execute and profile the `06_Threads` program.
- It will launch the vector_add kernel with 2^M threads
- Near linear parallel scaling inside the kernel



CUDA execution model - Threads

- Execute and profile the `06_Threads` program.
- It will launch the vector_add kernel with 2^M threads
- Amdahl's law hits
  - Near linear parallel scaling inside the kernel doesn't matter



06_Threads

Task:

- Run the code using the CUDA debugger – get some hands on
- Change #Threads from 1 to 1024
- Check the scaling using Nsight Systems

Link:

- https://classroom.github.com/a/mrB-yk8X

Goal:

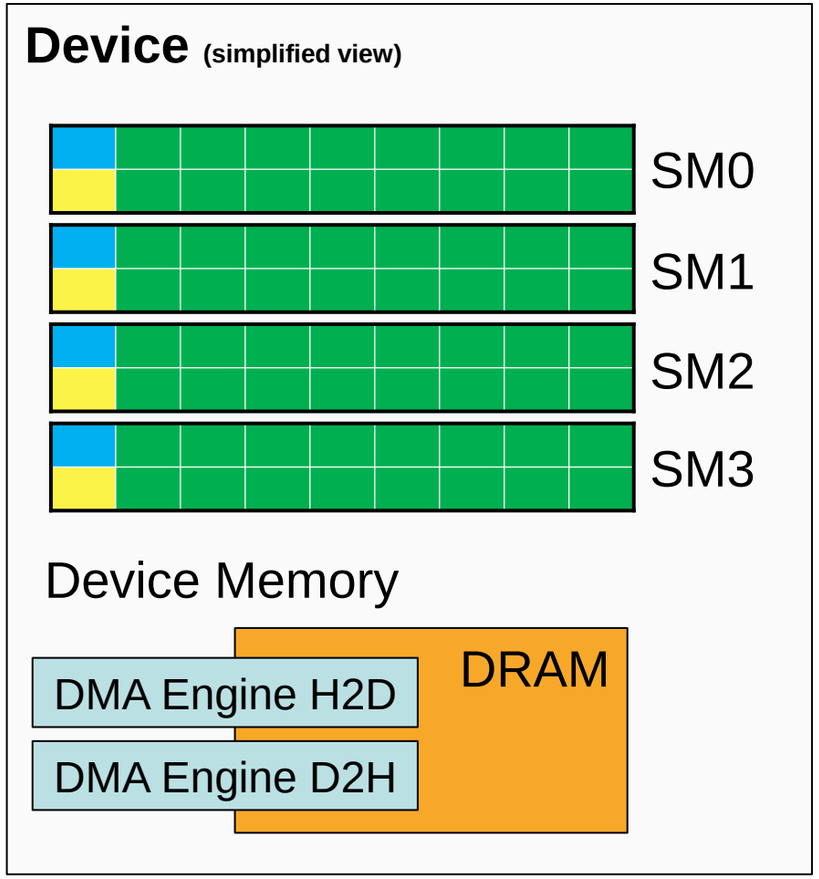
- Basic knowhow about the Nvidia tools
- Understand the basic scaling of #Threads used

CUDA execution model - Blocks

- A GPU has multiple Streaming Multiprocessors (SM)
- Each can run **multiple concurrent** thread blocks
- We define the thread blocks in a 1D, 2D or 3D grid where the dimensions are limited as
  $x \leq 2^{31}-1$ and $y, z \leq 65535$
- To utilize the GPU, we must create lots of thread blocks

block 0	block 1	block 2	block 3	send to SM0*
block 4	block 5	block 6	block 7	send to SM3*
block 8	block 9	block 10	block 11	send to SM2*
block 12	block 13	block 14	block 15	send to SM1*

* = no particular order or allocation rules, just an example



CUDA execution model - Blocks

- To identify the thread in the kernel code, use `threadIdx, blockIdx, blockDim and gridDim`
- Example `<<<dim3(4, 1, 1), dim3(8, 1, 1)>>>`

```
globalThreadID = blockIdx.x * blockDim.x + threadIdx.x
```

19 = 2 * 8 + 3

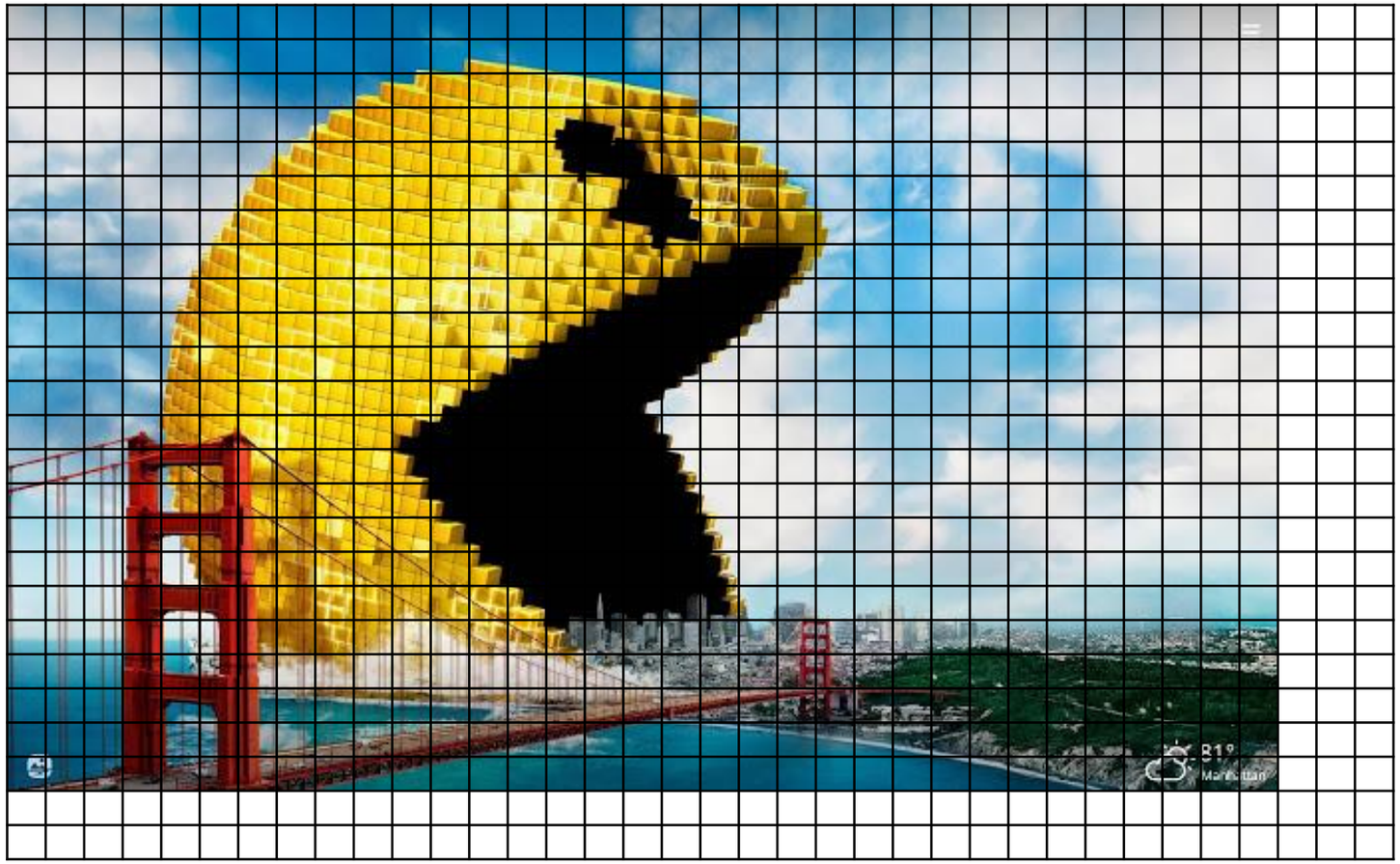
global thread ID (.x)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

threadIdx.x

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2								blockIdx.x = 3							

CUDA execution model - Blocks



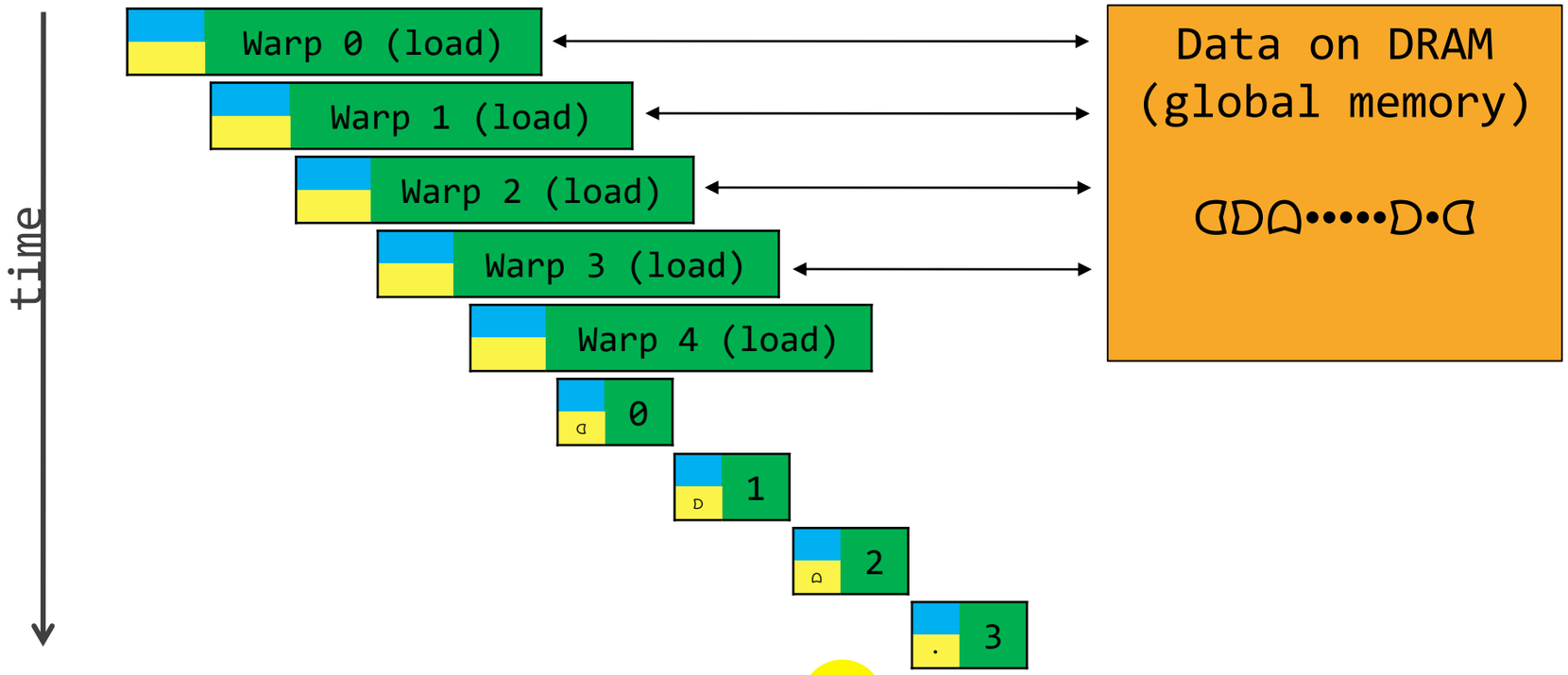
CUDA execution model

- Latency hiding is a major design principal of GPGPU
- Zero overhead thread scheduling
  - Tolerate long latency operations
  - Reason why GPUs have lower chip area for cache and branch prediction
    --> And more area for floating point execution

- Example: Hardware limits are 8 blocks/SM or 1024 Threads/SM and 512 Threads/block
  - 8x8 thread block: 64 Threads * 8 Blocks = 512 Threads on SM
  - 16x16 thread block: 256 Threads * 4 blocks = 1024 Threads on SM
  - 32x32 thread block: ~~1024 Threads~~ --> Too much for 1 block

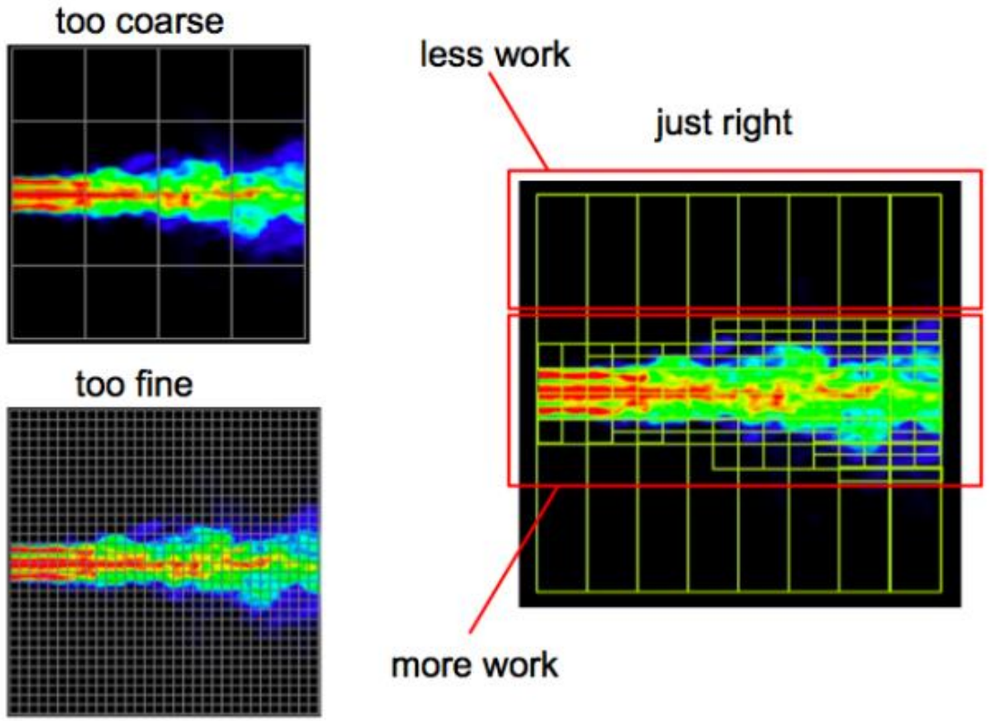
CUDA execution model - SIMD

- Latency hiding during parallel data loading
- The more warps (and thus threads) are in the game the higher the changes one has work to do



CUDA dynamic parallelism

- A fixed sized grid can lead to bad utilization of resources depending on the algorithm and used data
- Especially if we have nested parallelism:
  - Algorithms using recursion
  - Algorithms with independent batched work
  - Algorithms using hierarchical data structures
- CUDA allows to start new kernels from running kernels



Task vs Data parallelism

- GPUs are handling data parallelism very well
- The bigger the data size the more potential data parallelism is there
  - Free scaling with better hardware as more execution hardware is available
- Task parallelism is exposed through task decomposition of applications
  --> e.g. data flow graph
- We can also use Task parallelism within GPU code to enhance the parallelism
  --> e.g. CUDA streams (see in a few weeks)

Observations

- Use lots of threads inside a block to make use of latency hiding
- Use lots of thread blocks on the GPU to utilize all Streaming Multiprocessors
- `threadIdx, blockIdx, blockDim and gridDim` are used to identify a thread in the global scope
- Speedups in the kernel code alone do not necessarily increase the overall performance significantly --> Amdahl's Law

--> Think about 2D structures of threads and blocks. How do you do the mapping of the Ids?

06_Blocks

Task:

- Use `1<<20` Threads but with different grid and block sizes
- Implement the VectorAdd CUDA kernel using the theory of above
- Measure the execution speed and check out Nsight Compute and Nsight Systems
- Think about the reasons why the execution speed of the kernel differs?

Link:

- https://classroom.github.com/a/3hYUBx-_

Goal:

- Hands on CUDA code and Nvida Tools
- Understand the theory of above