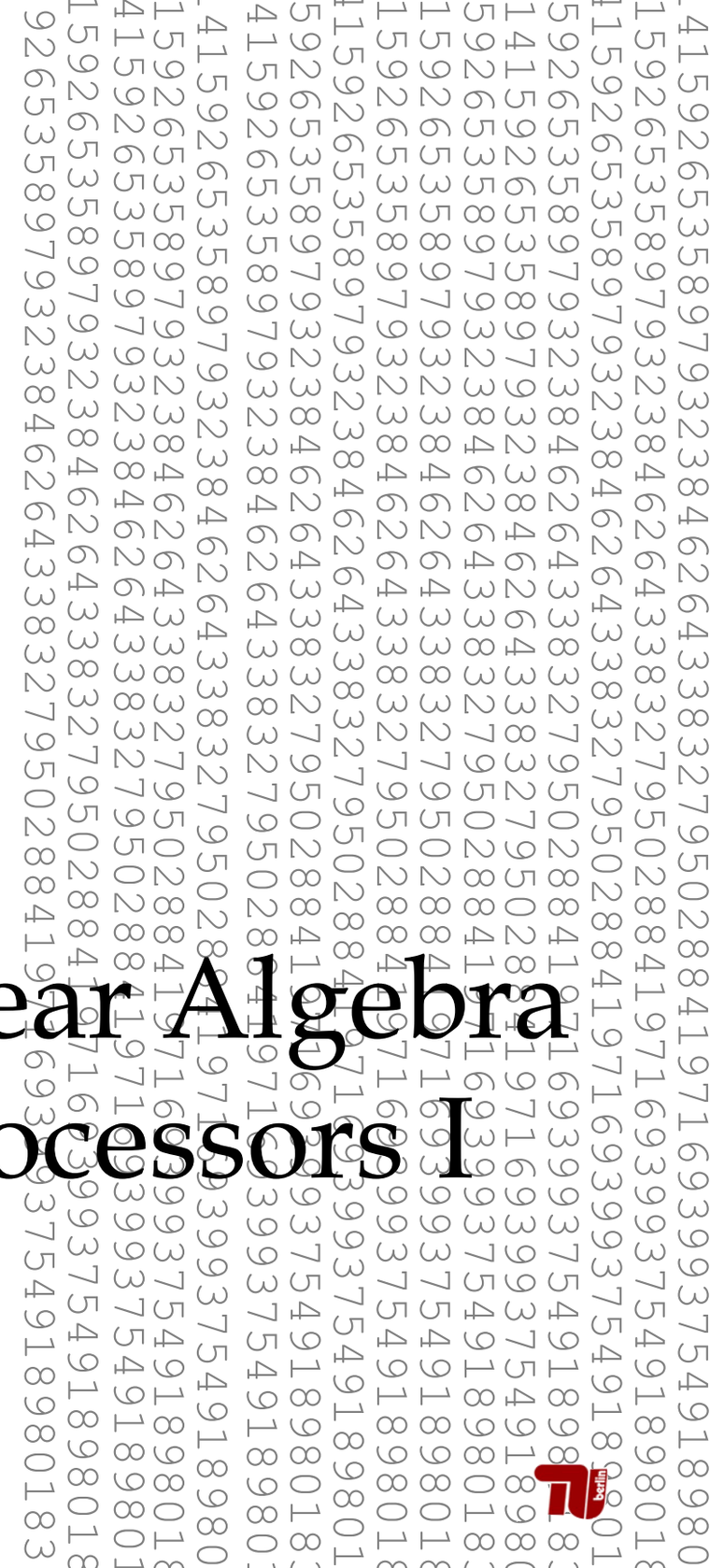
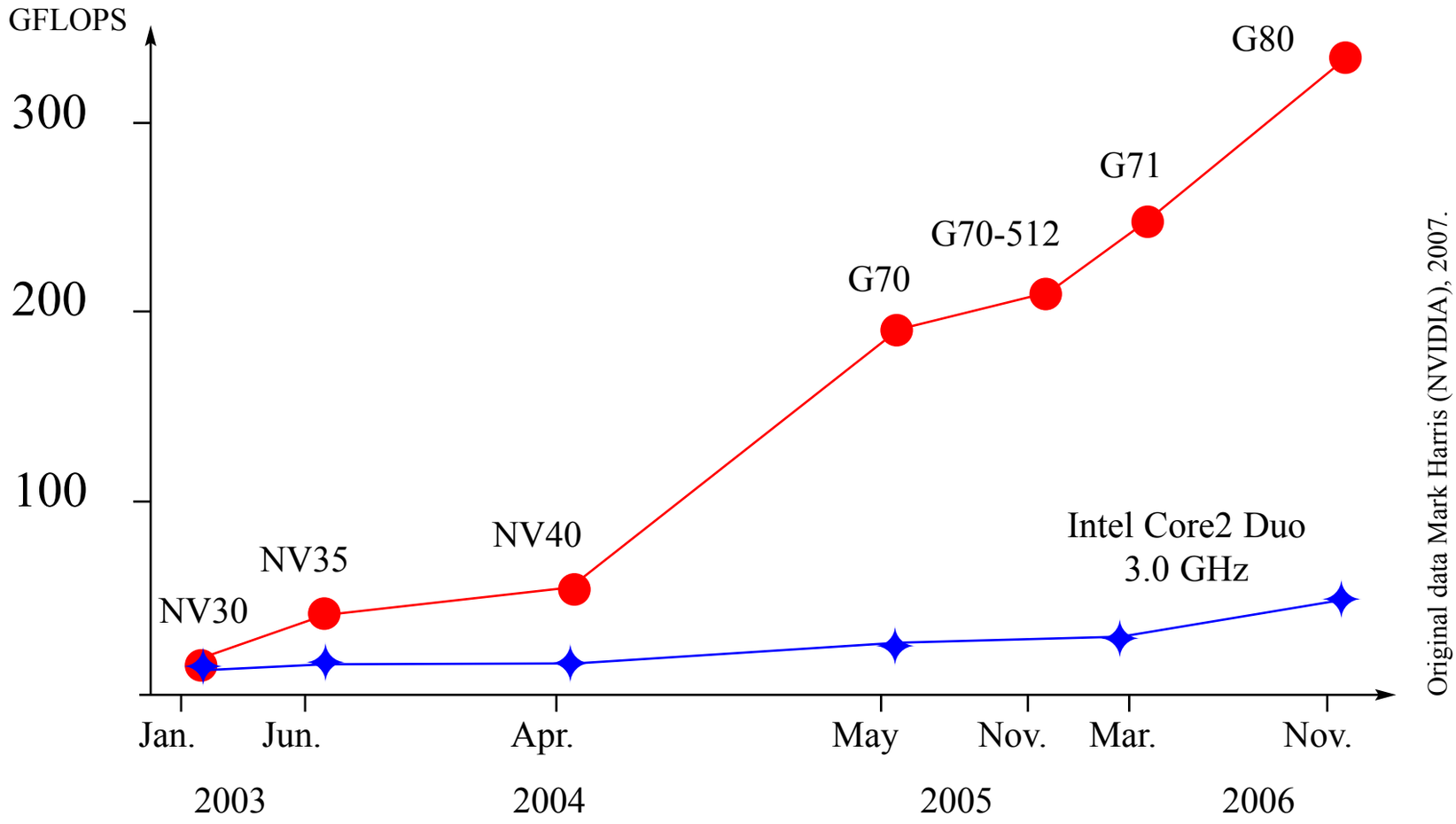
High Performance Linear Algebra on Data Parallel Co-Processors I

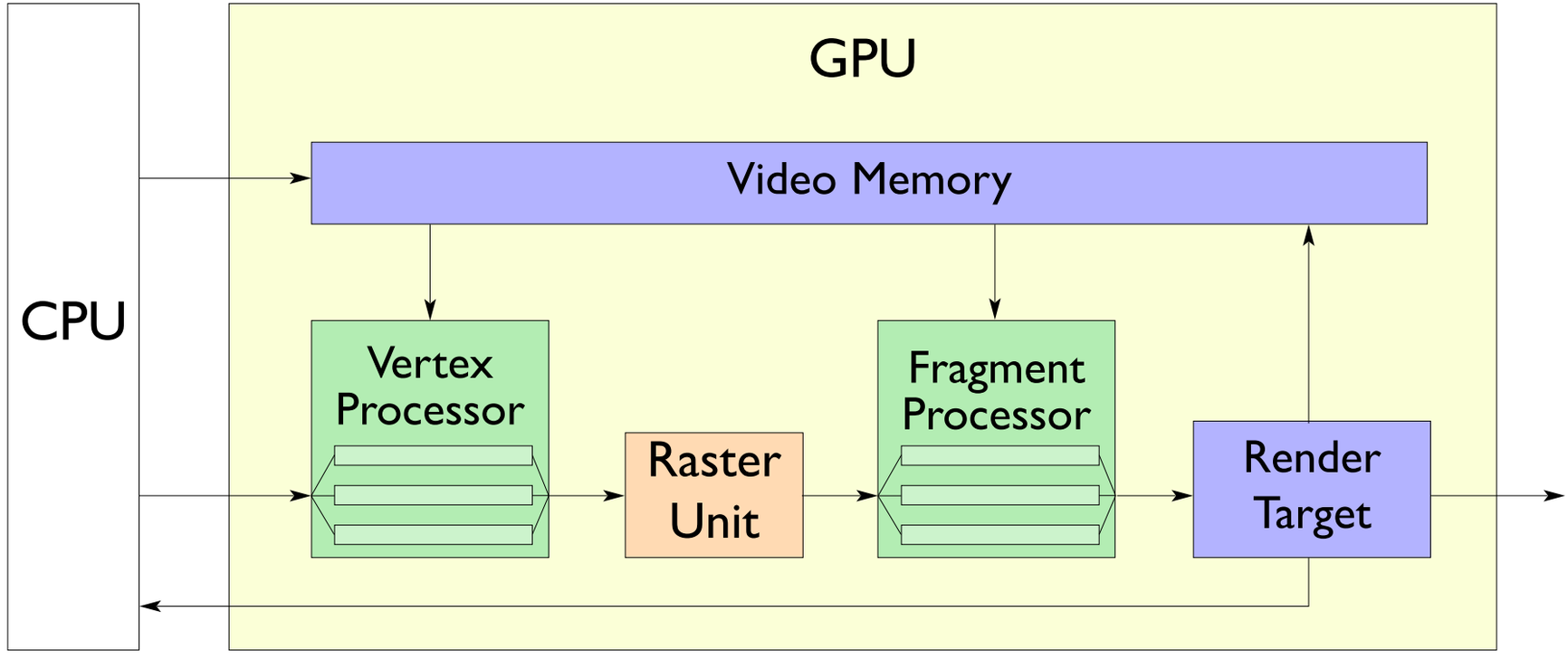


UNIVERSITY of TORONTO

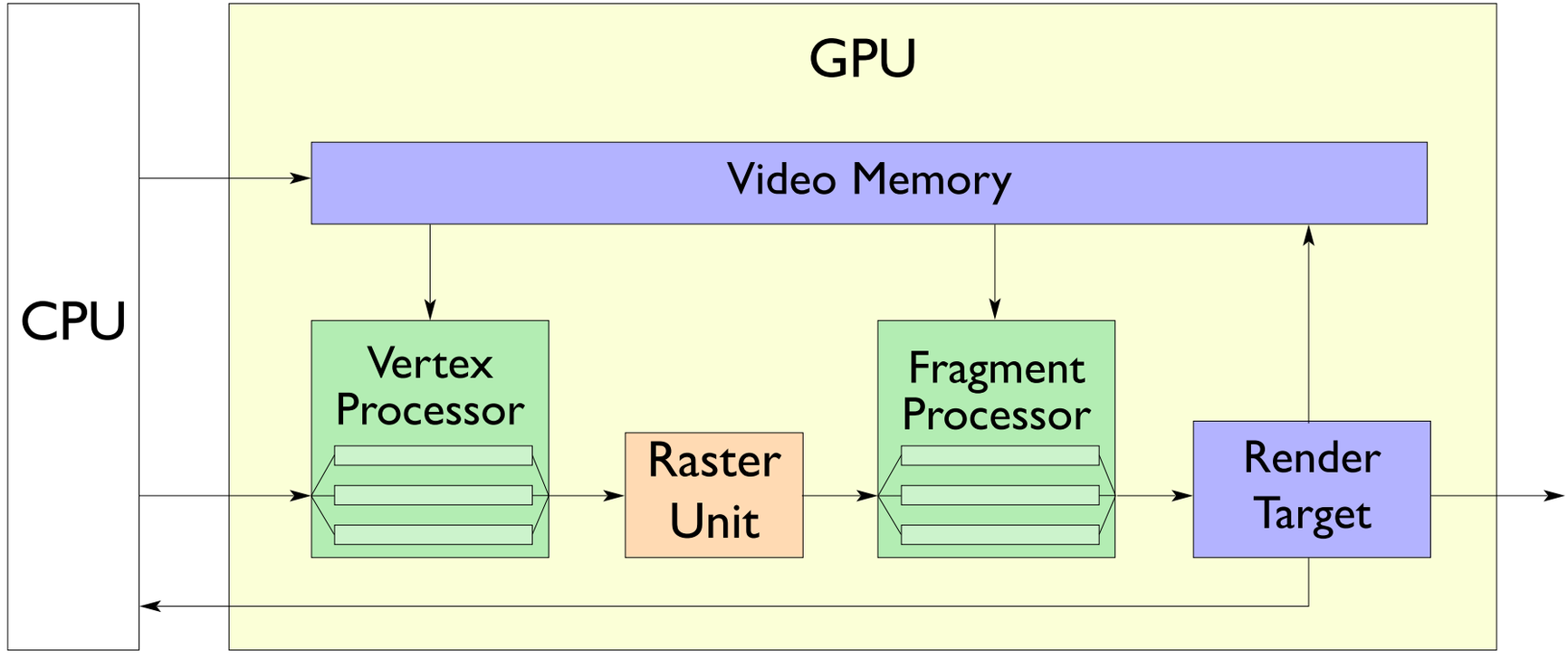
A bit of history

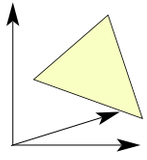


A bit of history

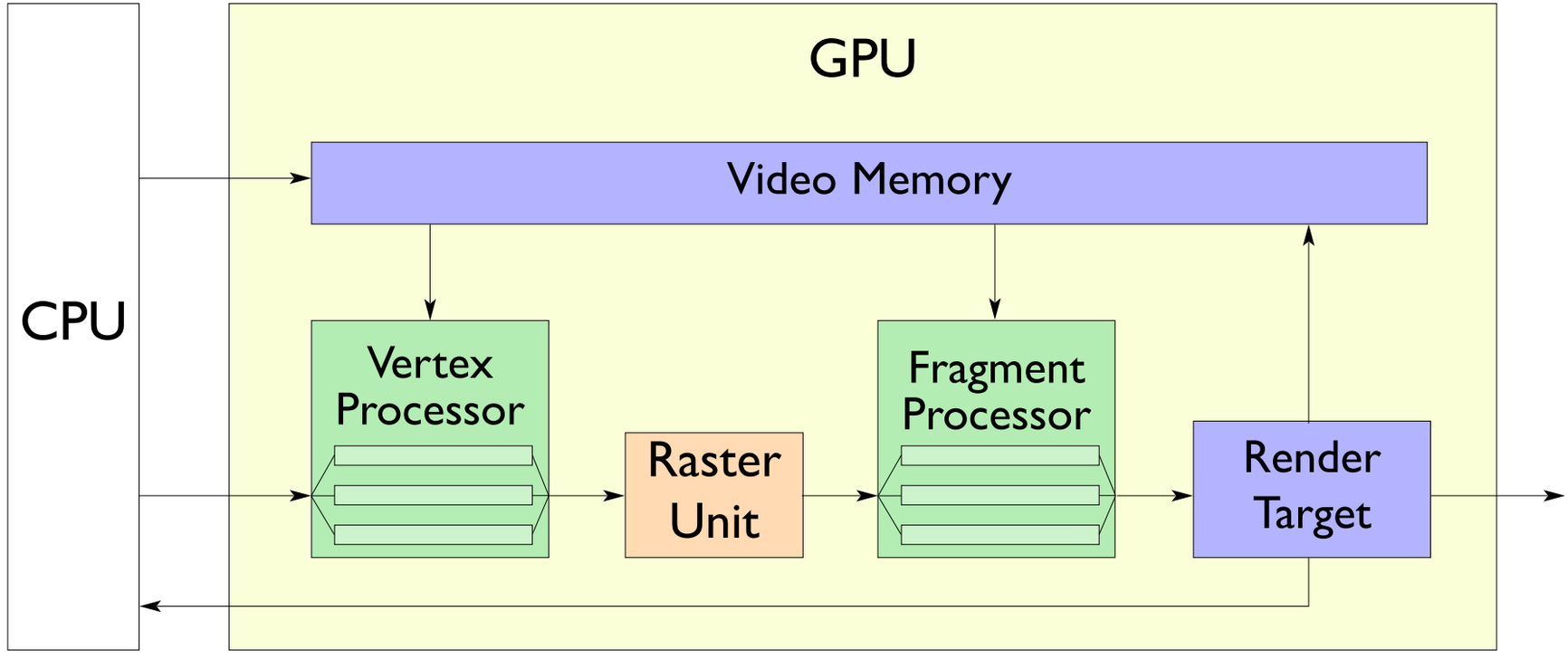


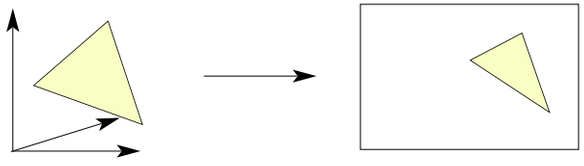
A bit of history



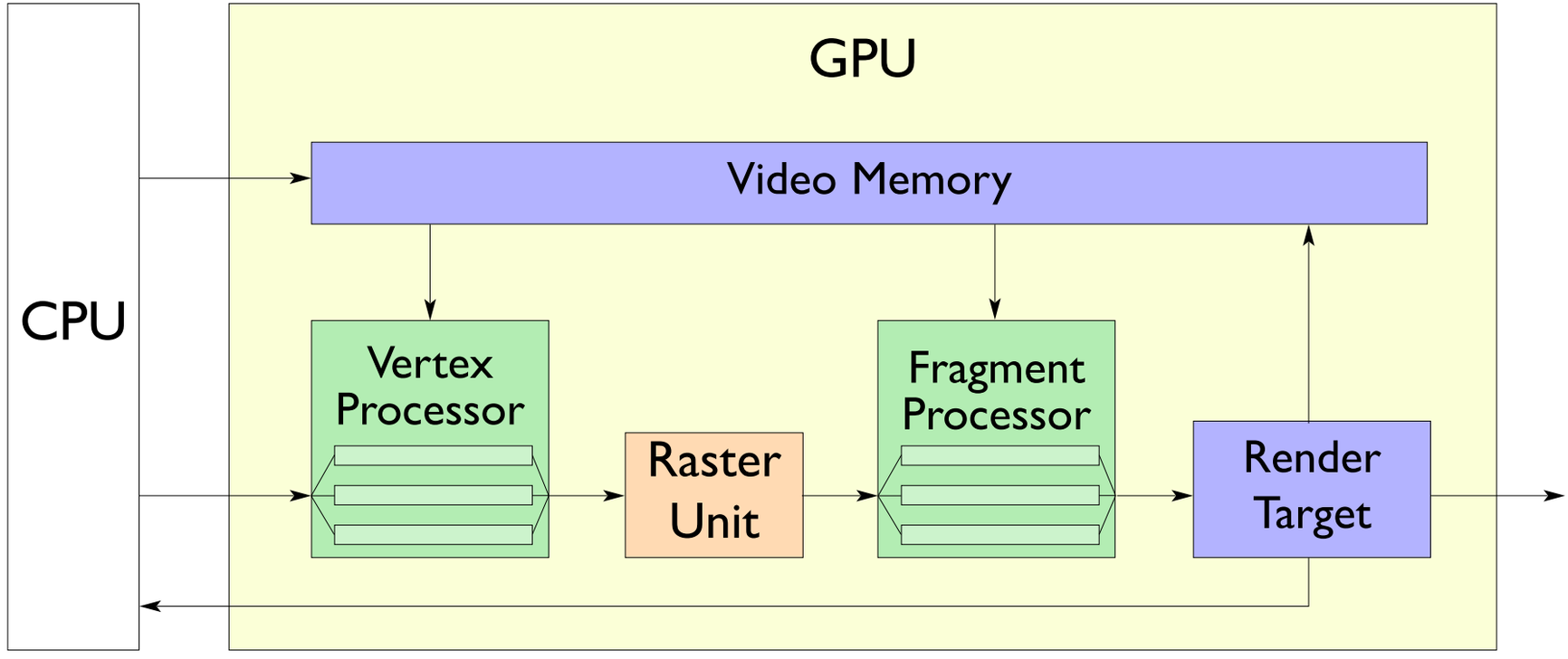


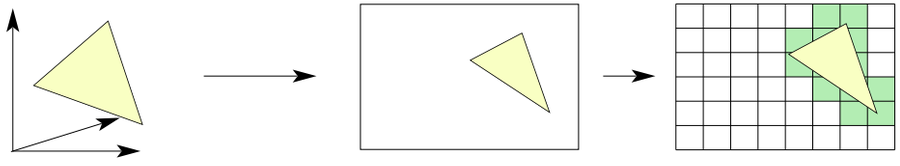
A bit of history



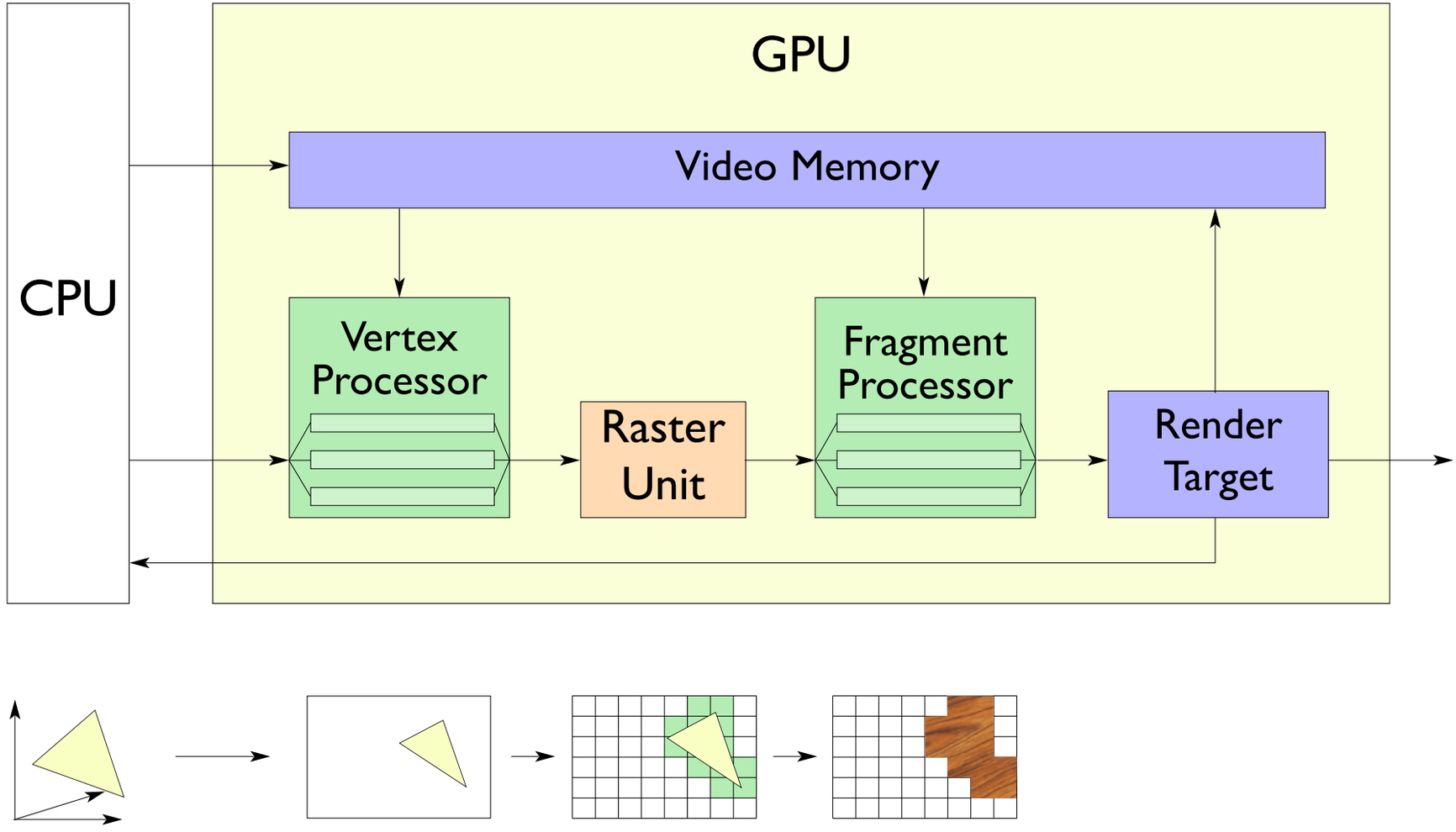


A bit of history

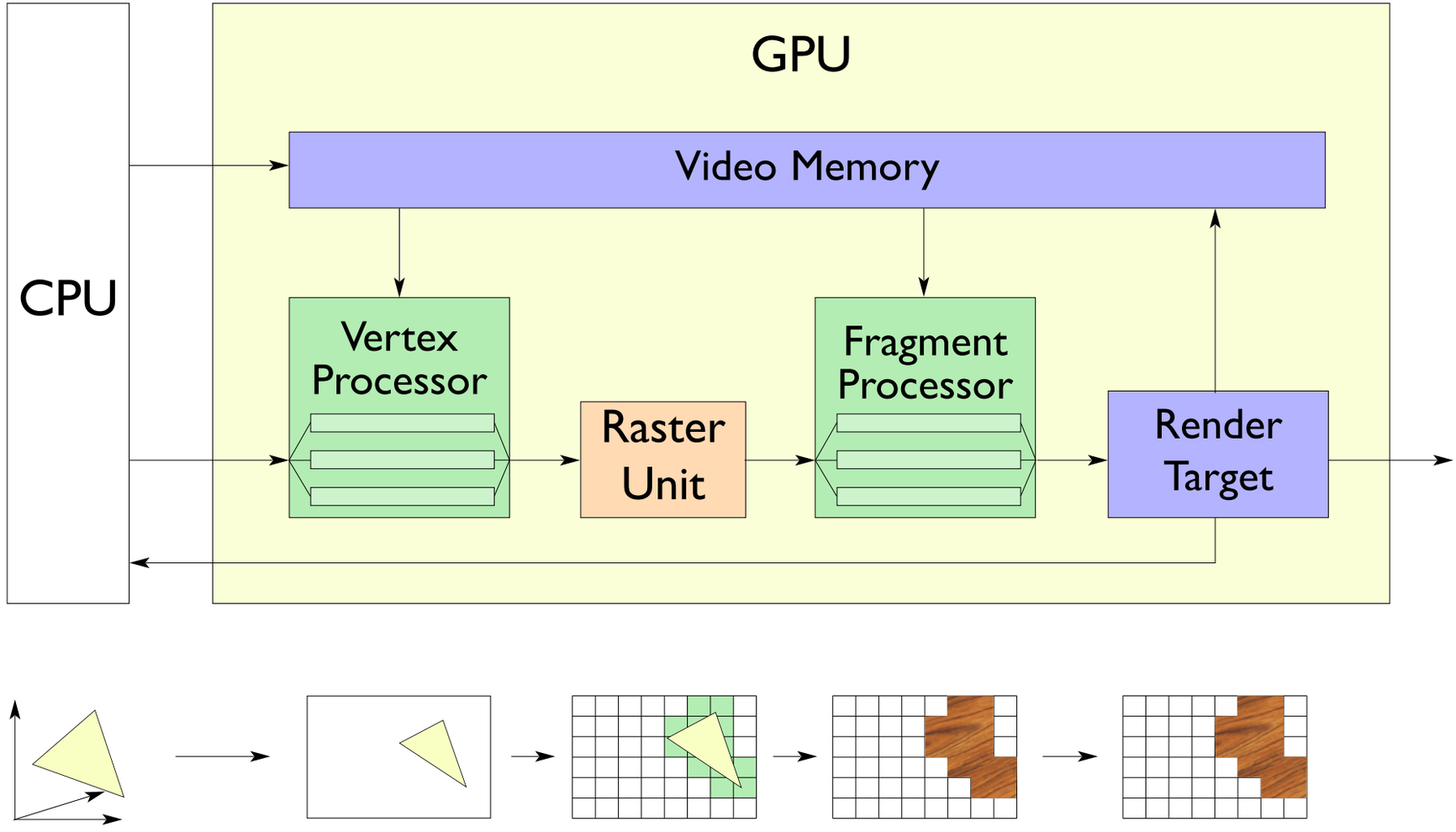




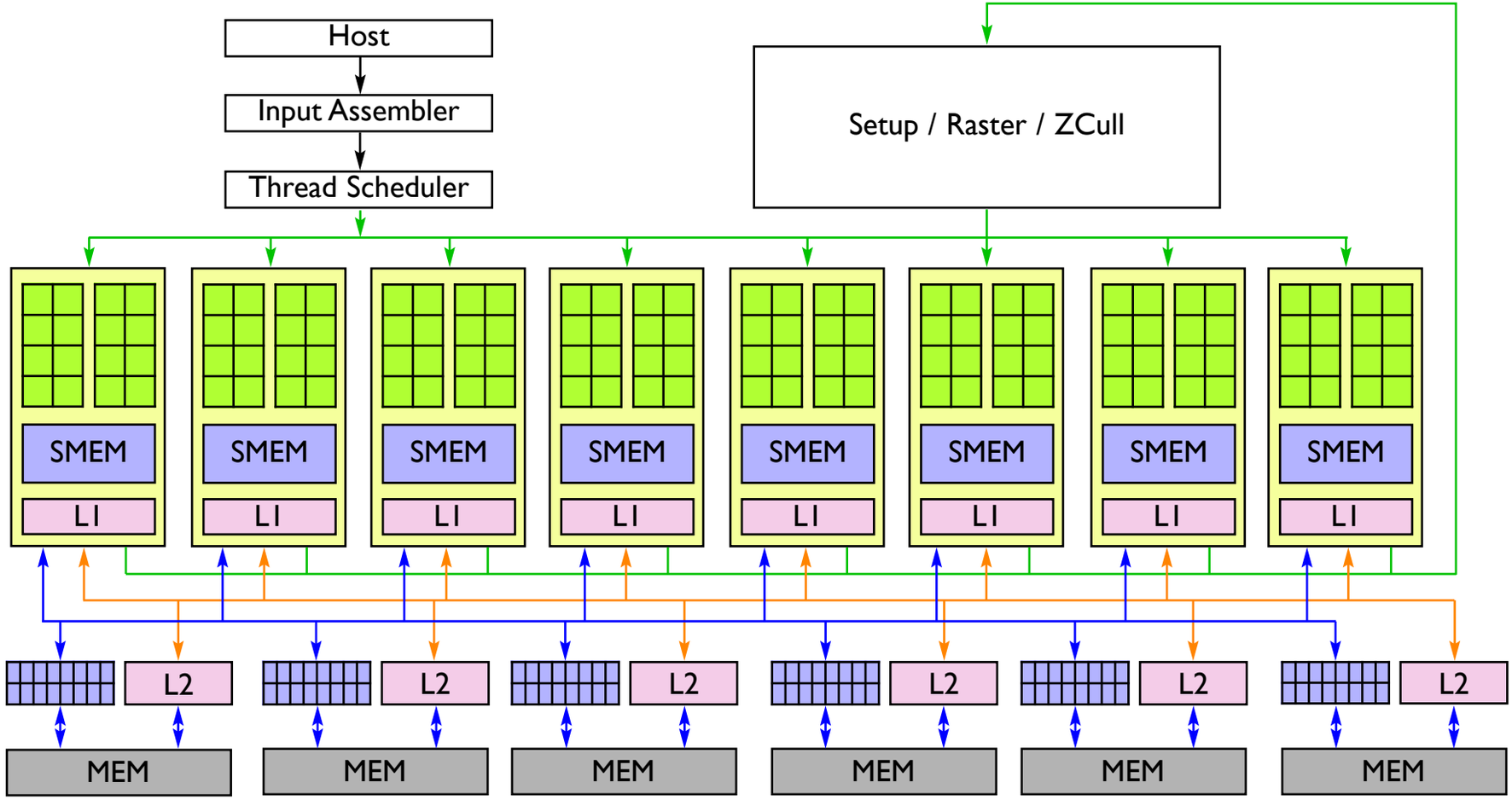
A bit of history



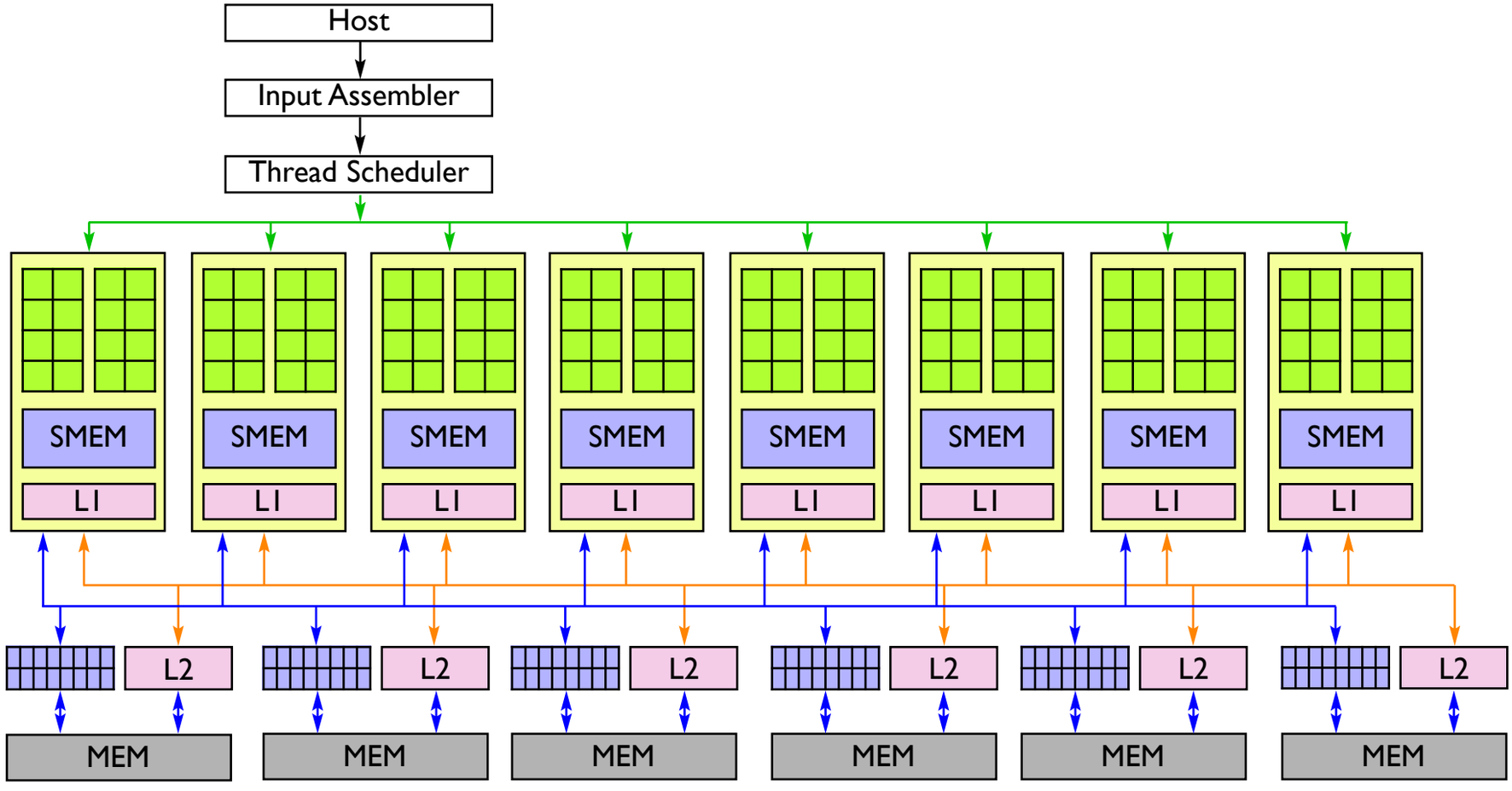
A bit of history



Modern graphics card



Data parallel co-processors



Array of multiprocessors

- Analogue of a multi-core chip with each multiprocessor corresponding to one core.
  - But the cores are vectorized.
- Task parallel execution.
  - Independent program counters.
- Between 2 and 16 multiprocessors on one chip.

Multiprocessor

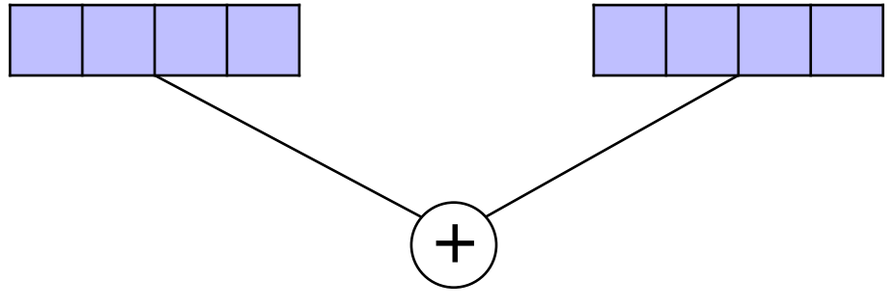
- Data parallel processing unit.

Multiprocessor

- Data parallel processing unit.

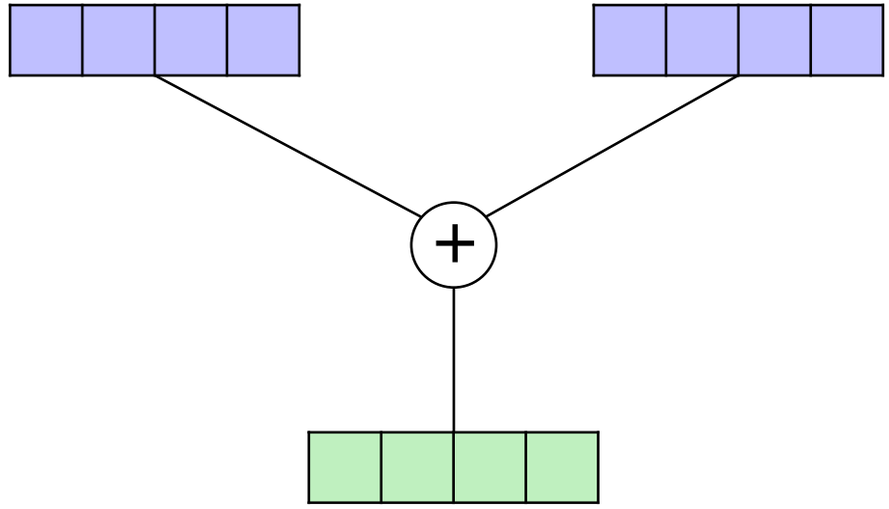
Multiprocessor

- Data parallel processing unit.



Multiprocessor

- Data parallel processing unit.



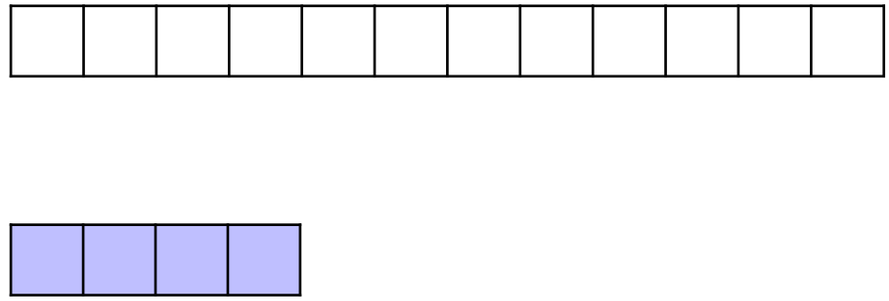
Multiprocessor

- Data parallel processing unit.
- **Thread block**: set of threads that is *logically* processed together on a multiprocessor.

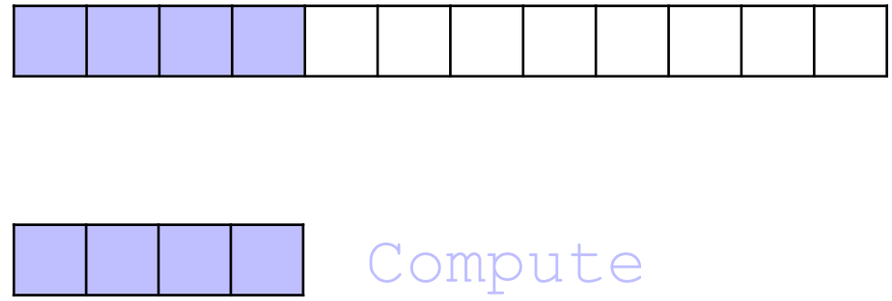
Multiprocessor

- Data parallel processing unit.
- **Thread block**: set of threads that is *logically* processed together on a multiprocessor.
- **Thread warp**: set of threads that is *physically* processed together on a multiprocessor.
  - Width of the processor, typically 16 or 32.

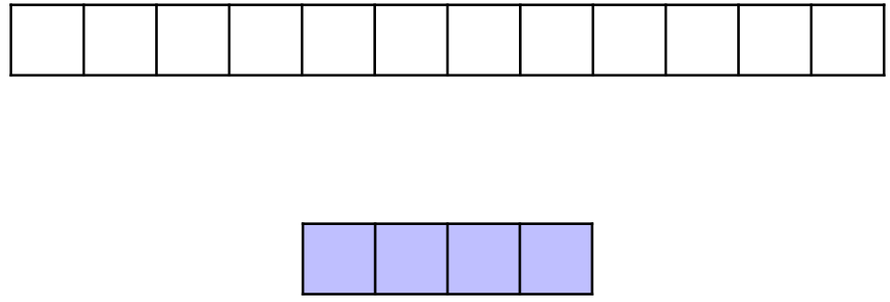
Multiprocessor



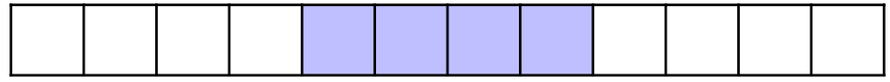
Multiprocessor



Multiprocessor

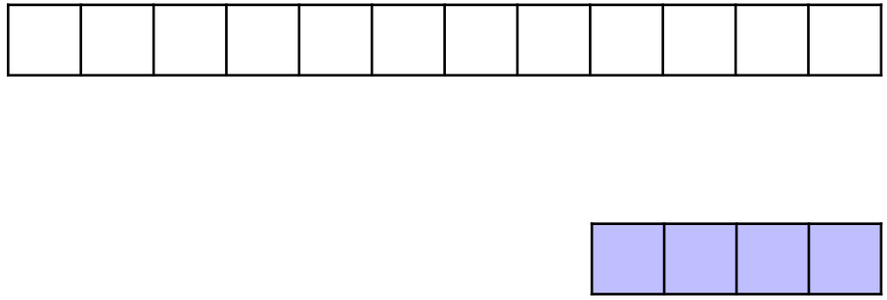


Multiprocessor

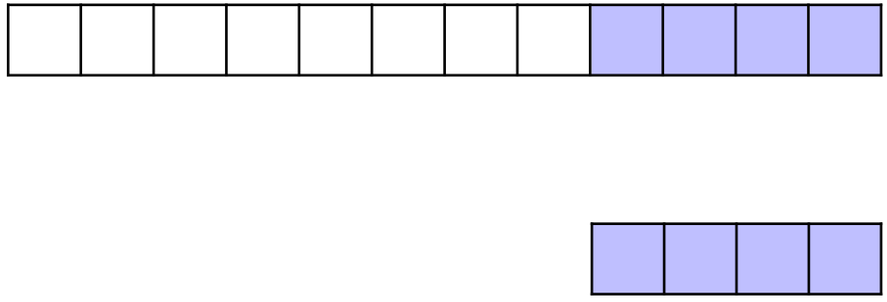


Compute

Multiprocessor



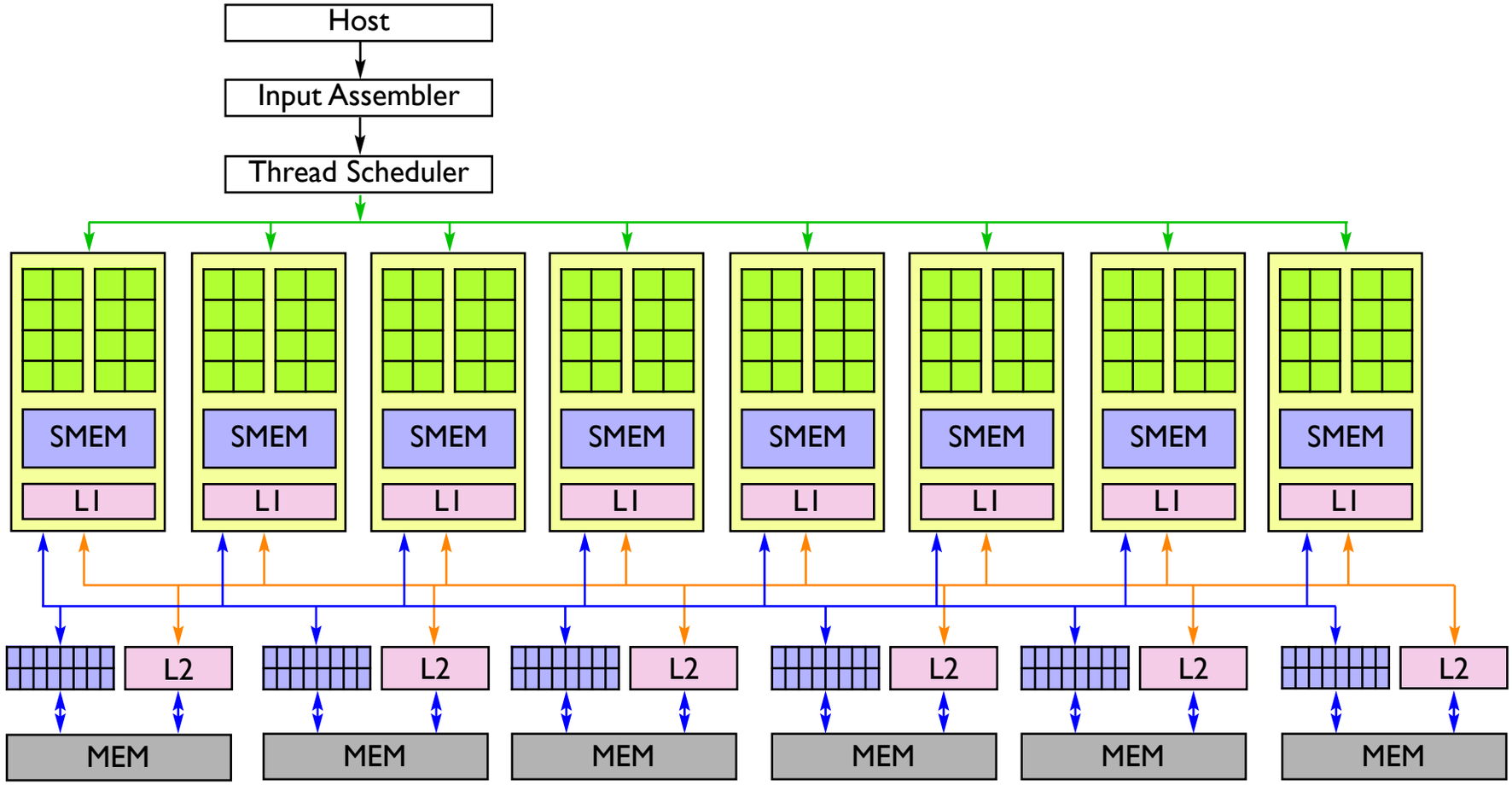
Multiprocessor



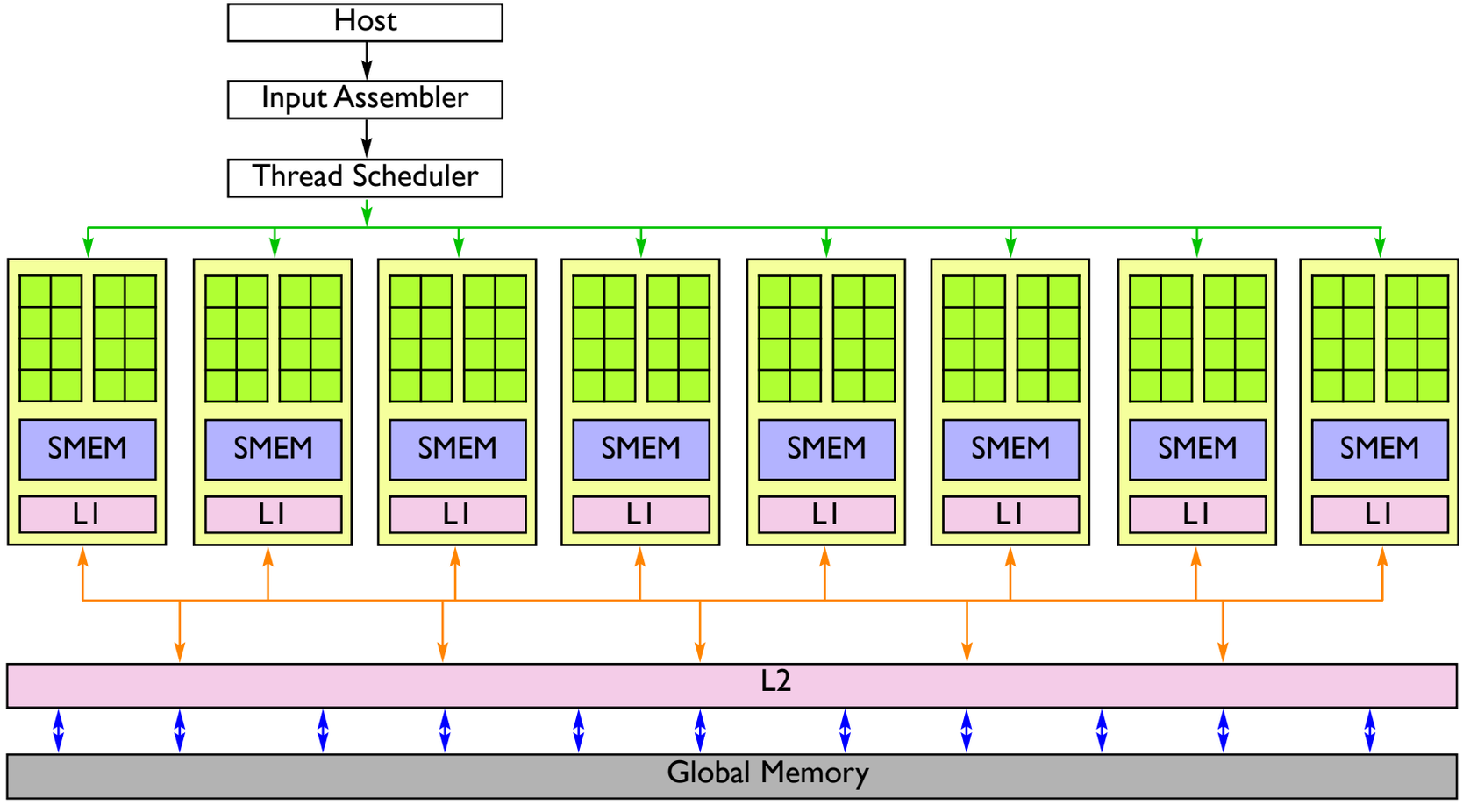
Multiprocessor

- Data parallel processing unit.
- **Thread block**: set of threads that is *logically* processed together on a multiprocessor.
- **Thread warp**: set of threads that is *physically* processed together on a multiprocessor.
  - Width of the processor, typically 16 or 32.
- **Thread**: processes one data element.
  - Extremely lightweight.

Data parallel co-processors



Data parallel co-processors



Memory vs. Execution

Task parallel array of multiprocessors	Global memory	Host and device very high latency

Memory vs. Execution

Task parallel array of multiprocessors	Global memory	Host and device very high latency
Thread block per multiprocessor	Local Memory (Shared Memory / L1)	Thread block only low latency

Memory vs. Execution

Task parallel array of multiprocessors	Global memory	Host and device very high latency
Thread block per multiprocessor	Local Memory (Shared Memory / L1)	Thread block only low latency
Warp of threads	–	–

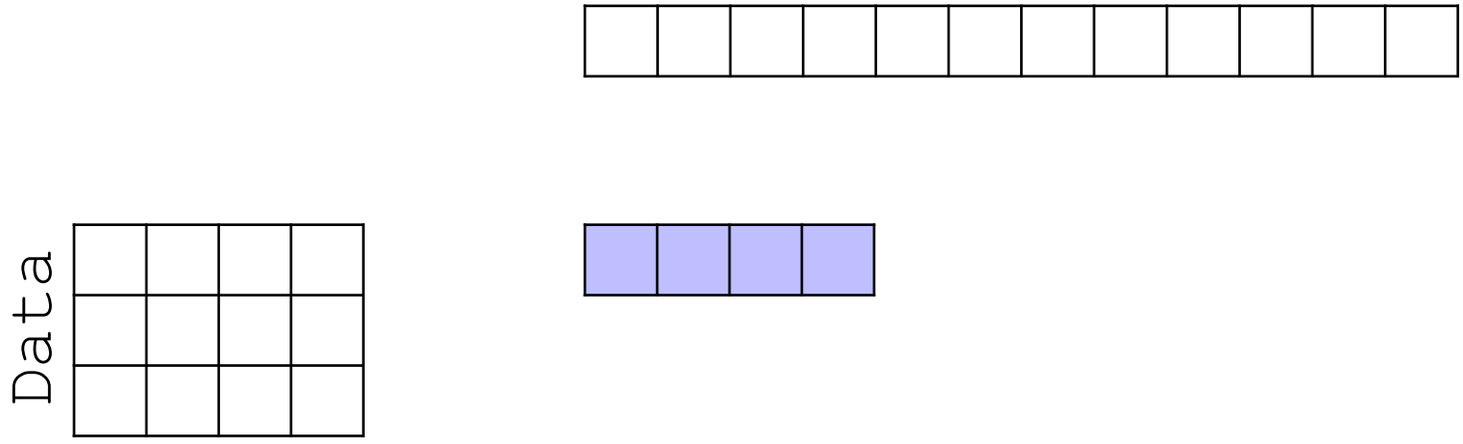
Memory vs. Execution

Task parallel array of multiprocessors	Global memory	Host and device very high latency
Thread block per multiprocessor	Local Memory (Shared Memory / L1)	Thread block only low latency
Warp of threads	–	–
Thread	Registers	Thread only very low latency

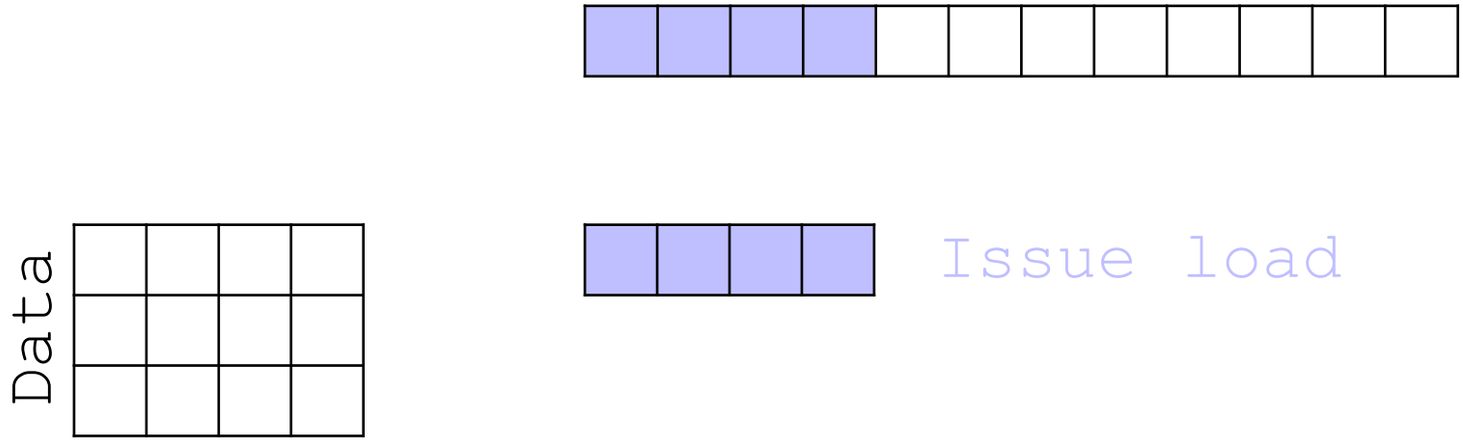
Memory vs. Execution

Task parallel array of multiprocessors	Global memory	Host and device very high latency
Thread block per multiprocessor	Local Memory (Shared Memory / L1)	Thread block only low latency
Warp of threads	–	–
Thread	Registers	Thread only very low latency

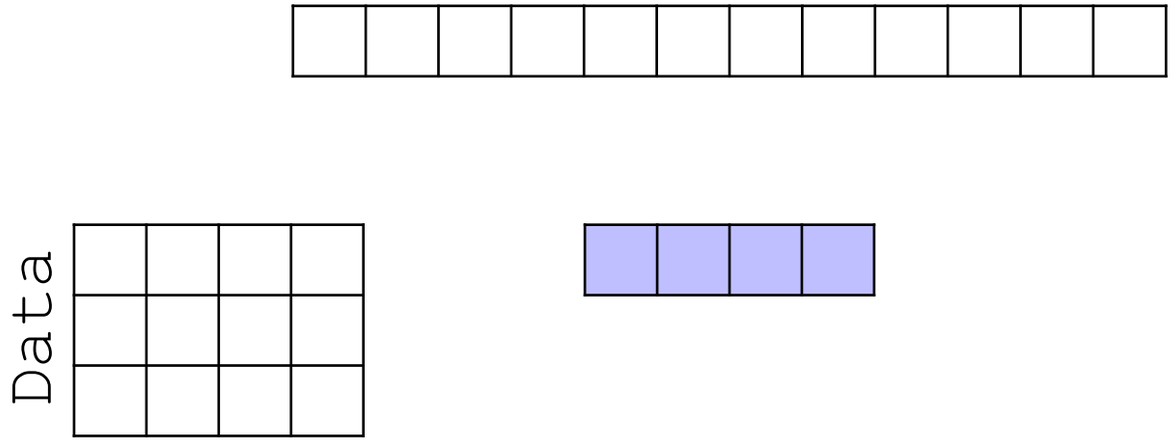
Thread level parallelism



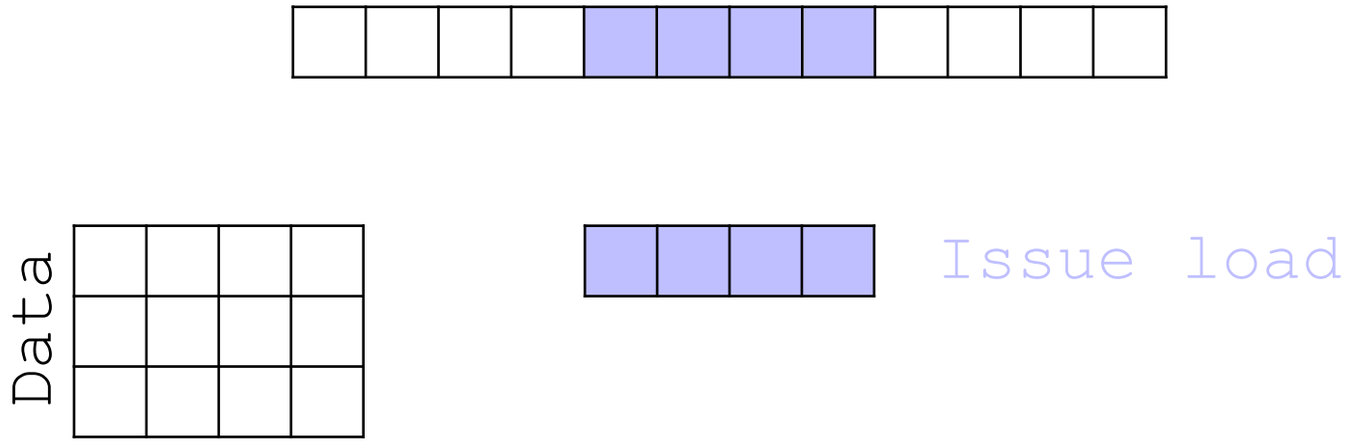
Thread level parallelism



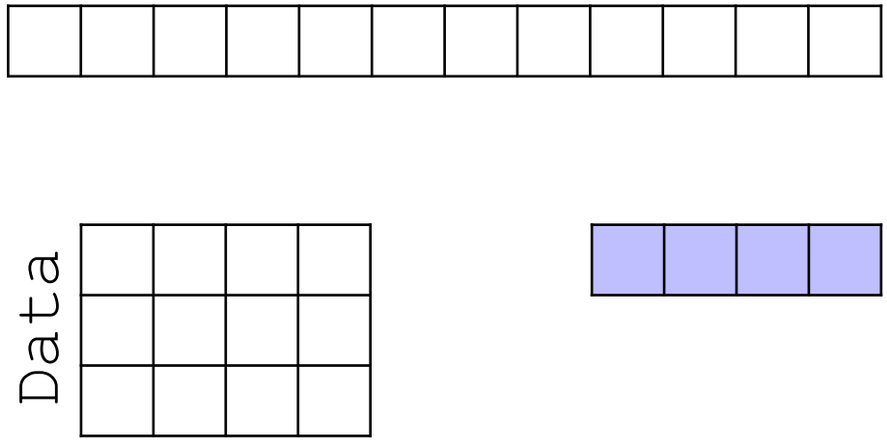
Thread level parallelism



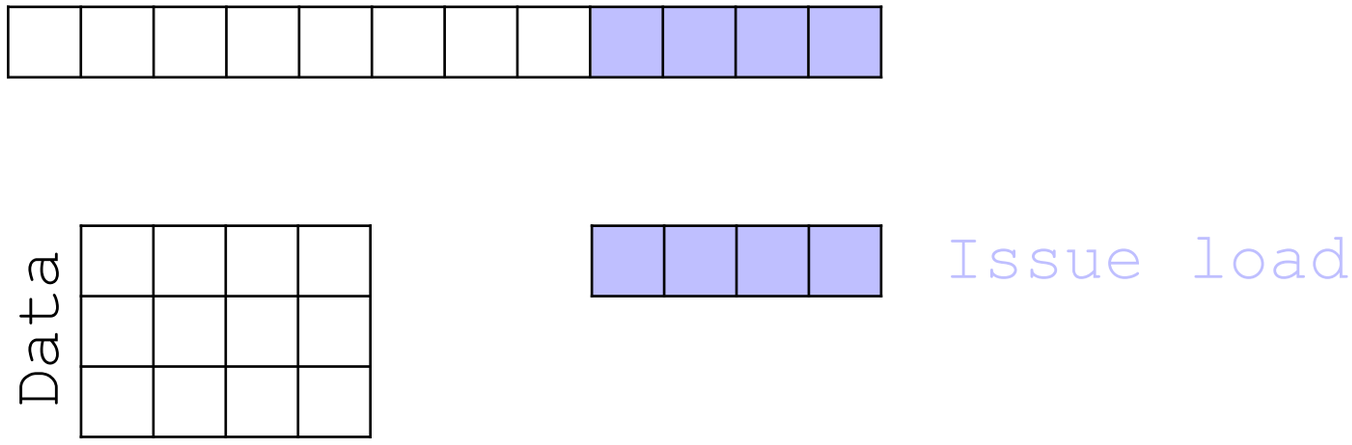
Thread level parallelism



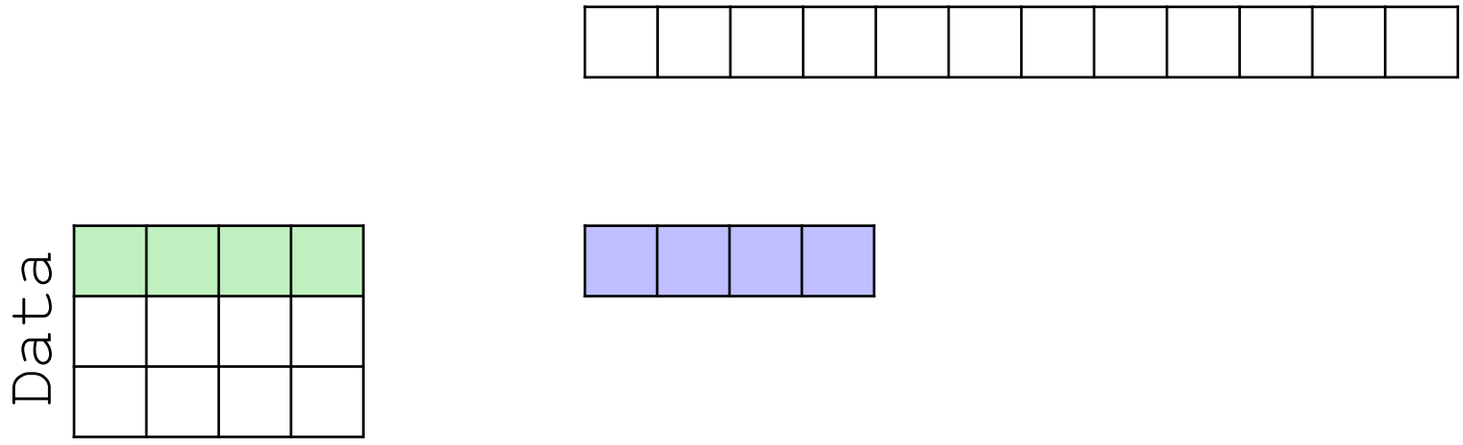
Thread level parallelism



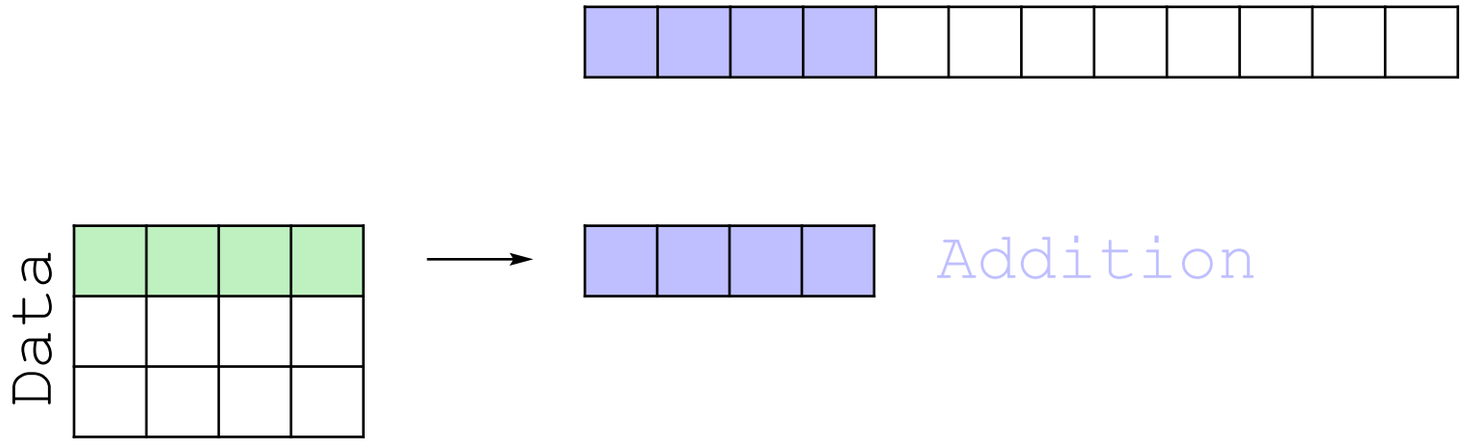
Thread level parallelism



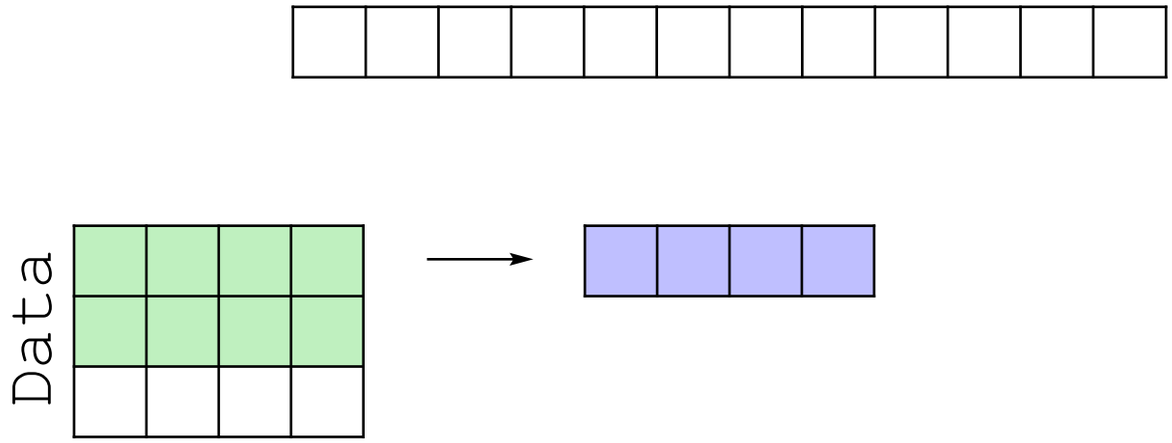
Thread level parallelism



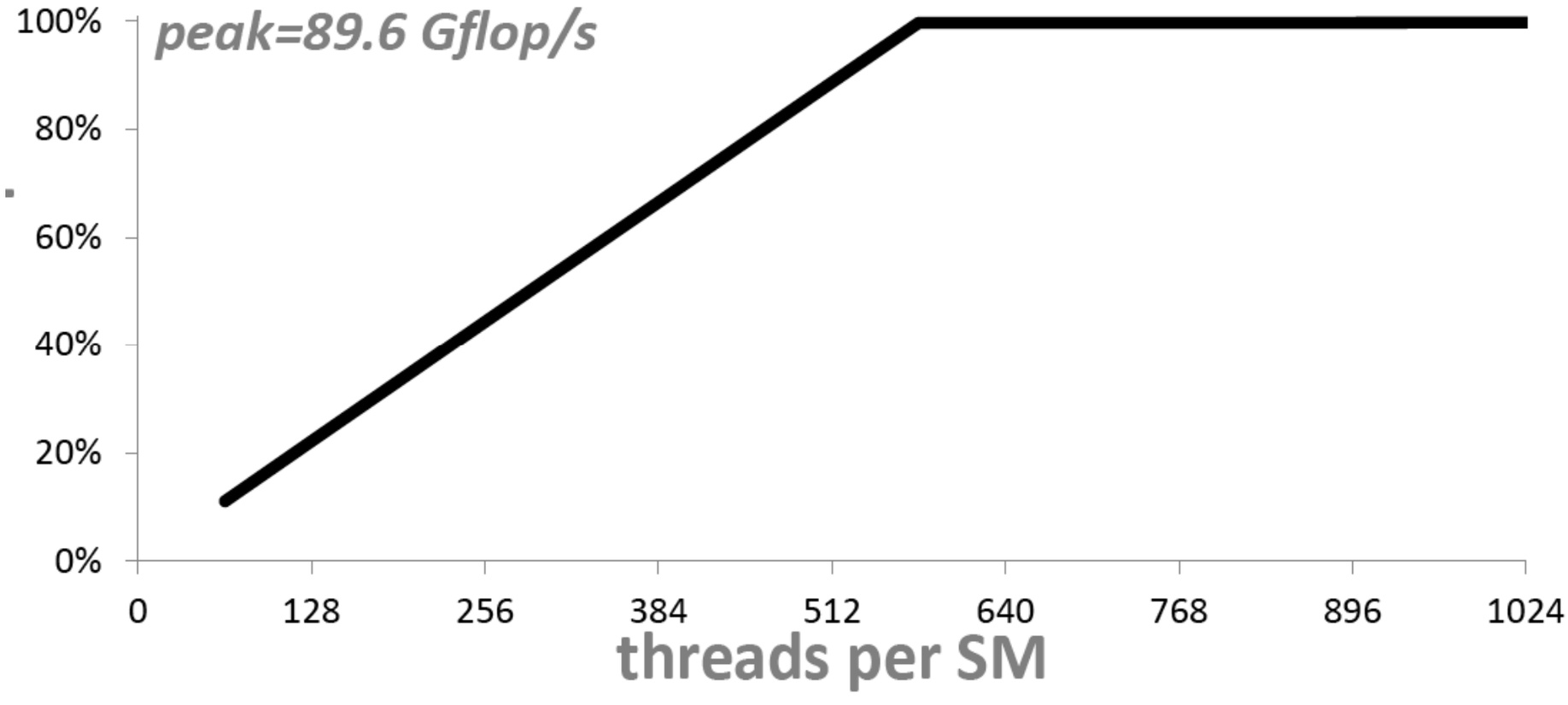
Thread level parallelism



Thread level parallelism



Thread level parallelism



http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf

Instruction Level Parallelism

```
...

c = c * b + a;
```

Instruction Level Parallelism

```
...

c = c * b + a;
d = d * b + a;
```

Instruction Level Parallelism

```
...

c = c * b + a;
d = d * b + a;
e = e * b + a;

...
```

Instruction Level Parallelism

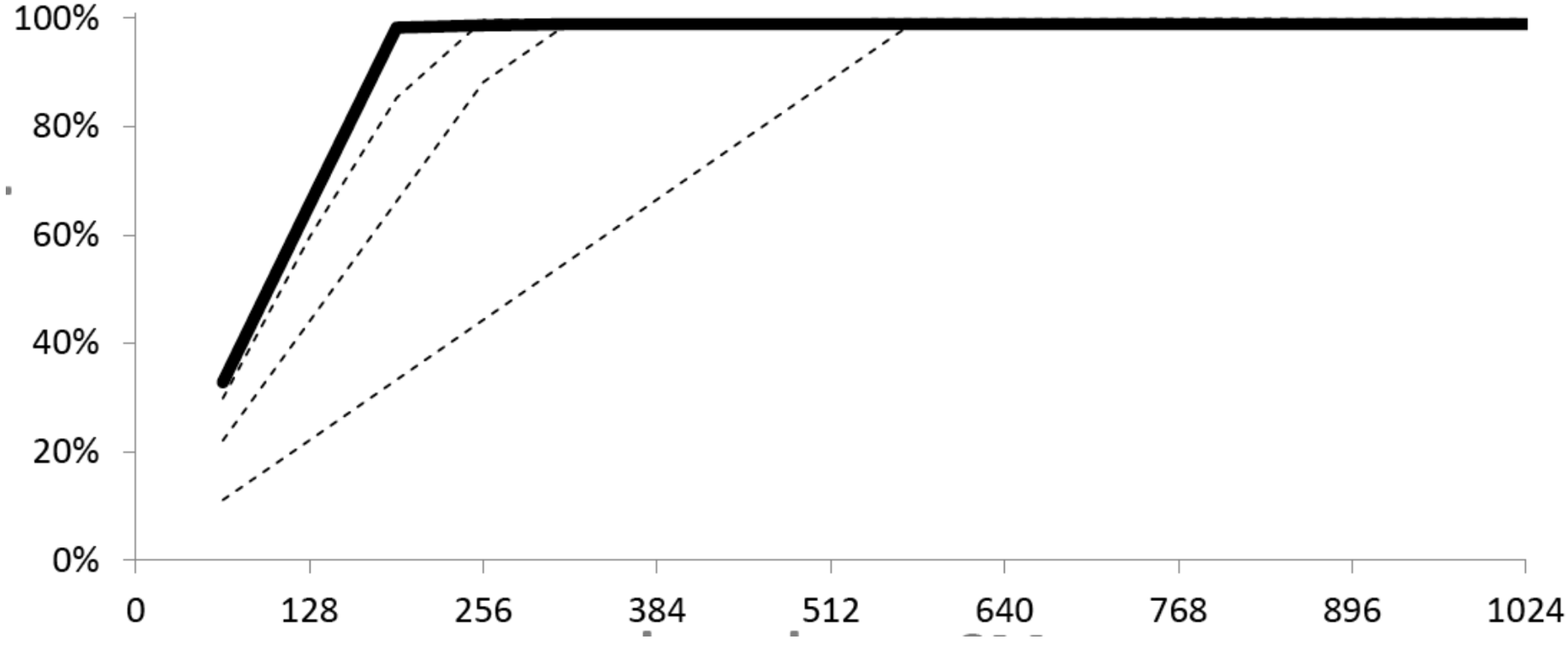
```
...

c = c * b + a;
d = d * b + a;
e = e * b + a;

...
```

Commands are non-blocking – only dependent operations block multiprocessor.

Instruction Level Parallelism



http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf

Memory latency

GPU	8800GTX	8600GTS	9800GTX	GTX280
at pins, GB/s	86	32	70	141
aligned copy	89%	83%	85%	89%
misaligned	9%	10%	9%	51%
stride-2	9%	10%	9%	45%
stride-10	10%	10%	9%	10%
stride-1000	0.9%	2.1%	1.1%	1.1%

Volkov, V., and J. W. Demmel. *Benchmarking GPUs to Tune Dense Linear Algebra*. 2008.

Memory latency

GPU	8800GTX	8600GTS	9800GTX	GTX280
at pins, GB/s	86	32	70	141
aligned copy	89%	83%	85%	89%
misaligned	9%	10%	9%	51%
stride-2	9%	10%	9%	45%
stride-10	10%	10%	9%	10%
stride-1000	0.9%	2.1%	1.1%	1.1%

Volkov, V., and J. W. Demmel. *Benchmarking GPUs to Tune Dense Linear Algebra*. 2008.

Ideal program flow

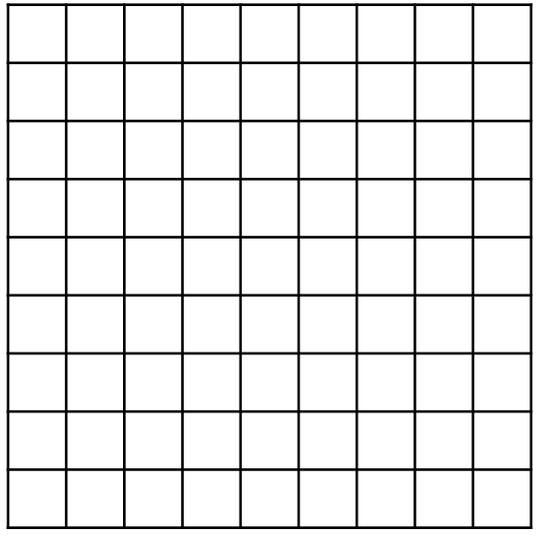
1. Load data form global to shared memory.
2. Process data in shared memory.
3. Write data from shared to global memory.

CUDA

- **C**ompute **U**nified **D**evice **A**rchitecture.
- Proprietary solution by NVIDIA.
  - But very similar to OpenCL.
- Low-level software interface based on ANSI C.
  - Explicit parallelization, memory handling, ...
  - Latest hardware has full C++ support.

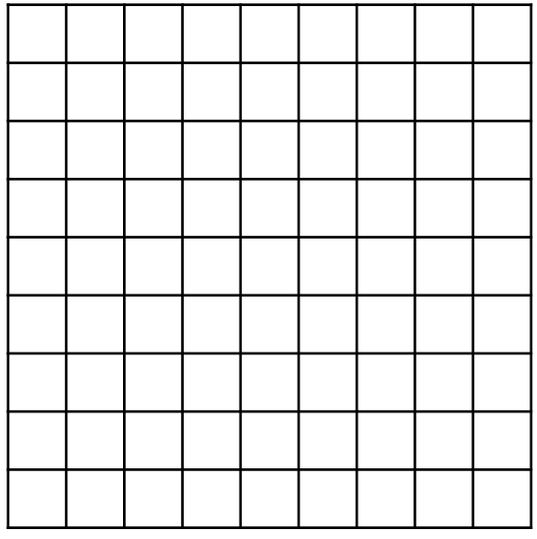
CUDA execution model

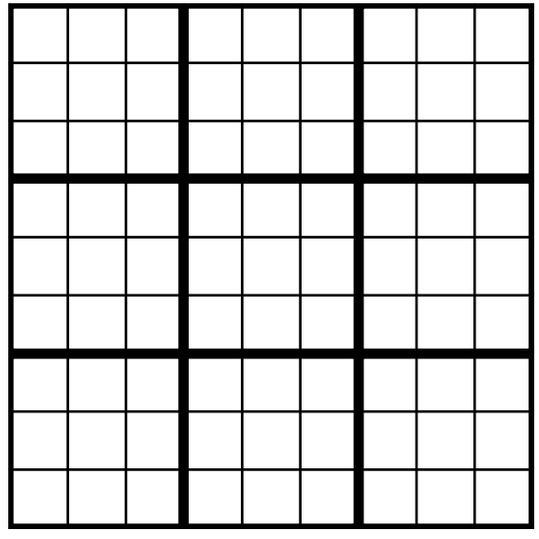
- A program is executed as **grid of thread blocks**.
- Each thread block and thread has unique {1,2,3}D **thread block and thread identifier**.



CUDA execution model

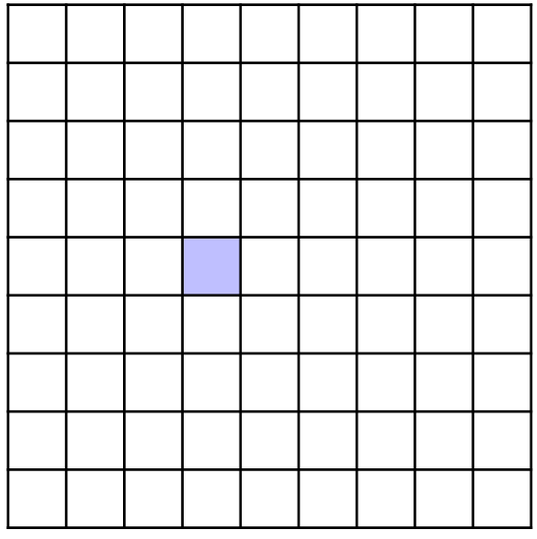
- A program is executed as **grid of thread blocks**.
- Each thread block and thread has unique {1,2,3}D **thread block and thread identifier**.

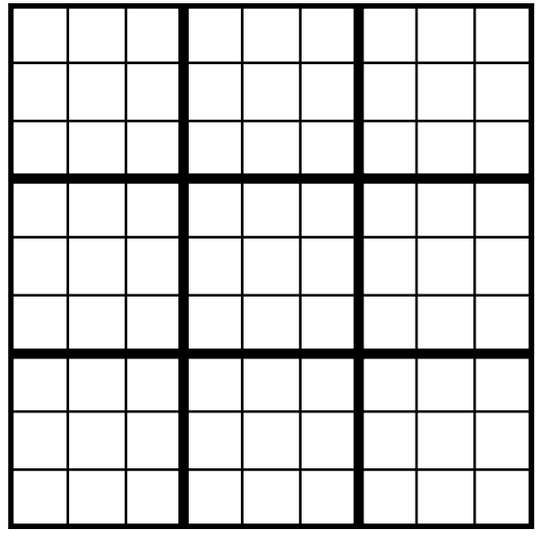




CUDA execution model

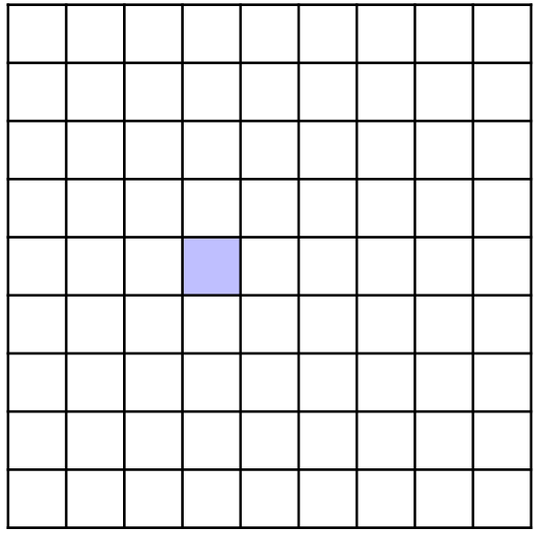
- A program is executed as **grid of thread blocks**.
- Each thread block and thread has unique {1,2,3}D **thread block and thread identifier**.

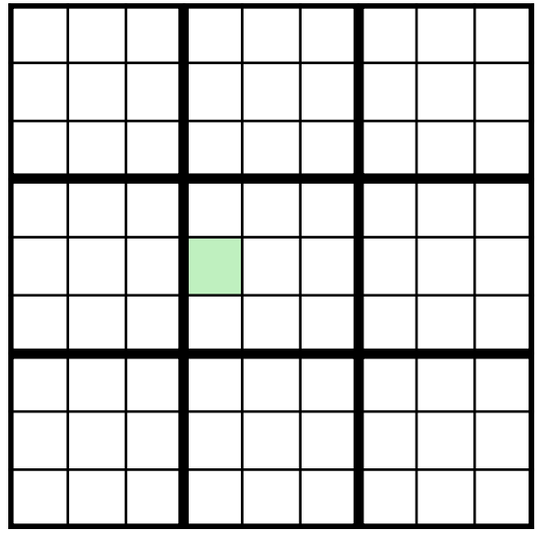




CUDA execution model

- A program is executed as **grid of thread blocks**.
- Each thread block and thread has unique {1,2,3}D **thread block and thread identifier**.





CUDA program structure

- **Kernel**: function executed on the device that can be called from the host.
- **Host program**: Device and memory initialization and execution management.
  - C / C++ / Fortran program.

CUDA Host Extensions

- Kernel invocation

```
myKernel<<< grid_dim , block_dim >>>(in, out)
```

- Memory management

```
cudaMalloc(), cudaMemcpy(), cudaFree(), ...
```

- Device management

```
cudaGetDeviceCount(), cudaSetDevice(), ...
```

- ...

CUDA Device Extensions

- Memory declaration

  ```
  __shared__ float smem[1024]
  ```

- Synchronisation (barrier)

  ```
  __synchthreads()
  ```

- Atomic operations

  ```
  atomicAdd(), atomicExch(), atomicXor(), ...
  ```

- Thread identifier

  ```
  threadIdx, blockIdx, blockDim, ...
  ```

CUDA Program Flow

1. Upload data to process into device memory.
2. Define execution environment.
3. Launch kernel.
   - Read input data into shared memory.
   - Process data.
   - Write result from shared to global memory.
4. Read result back to host memory.

CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

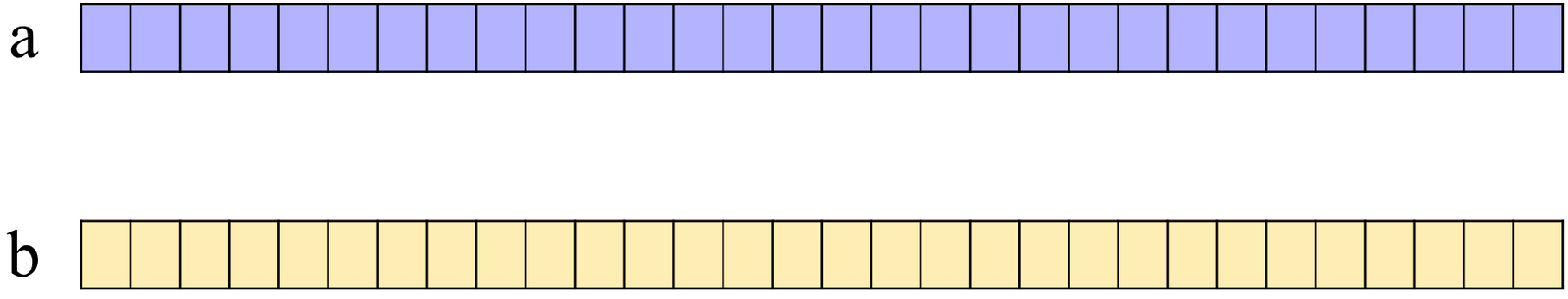
- Assume infinite signals and ignore necessary block overlap

CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap

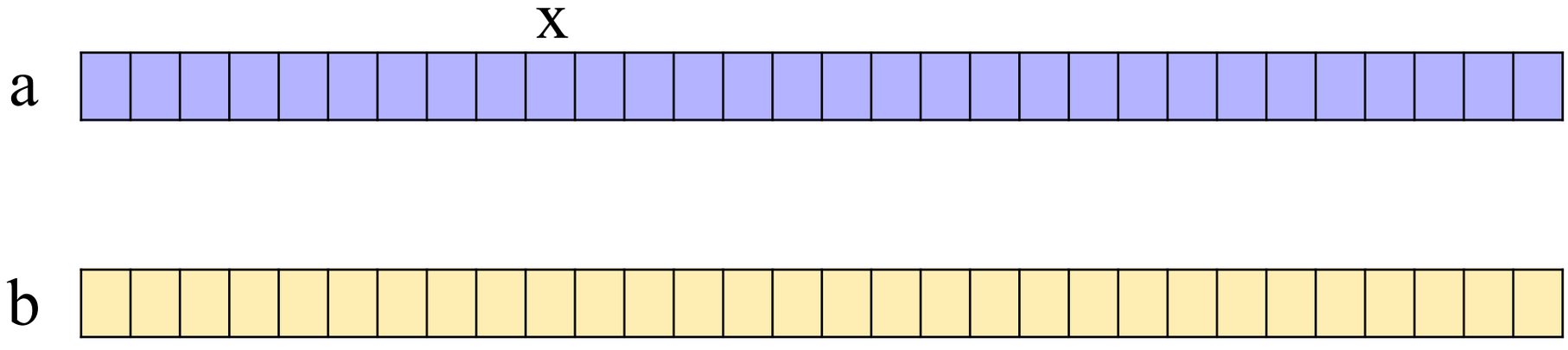


CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap

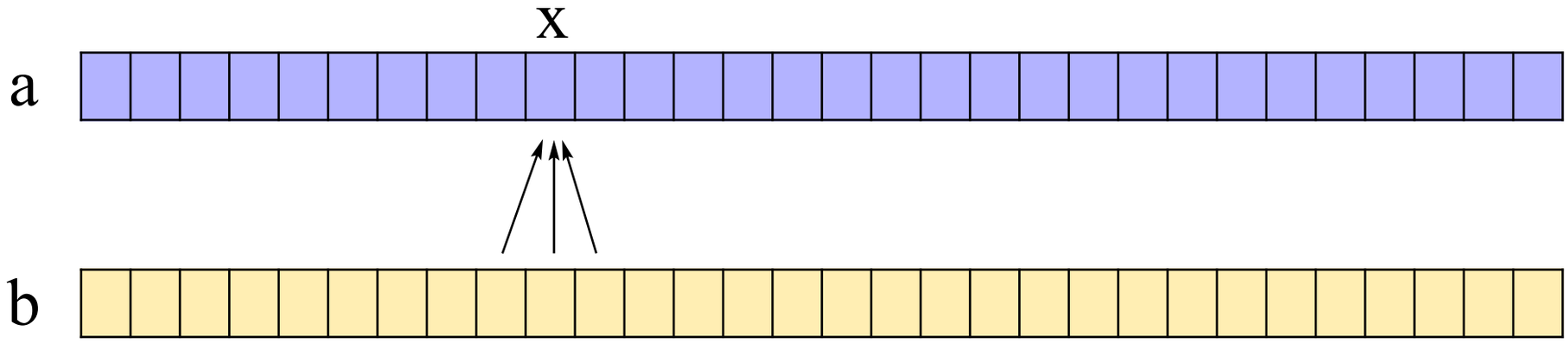


CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap

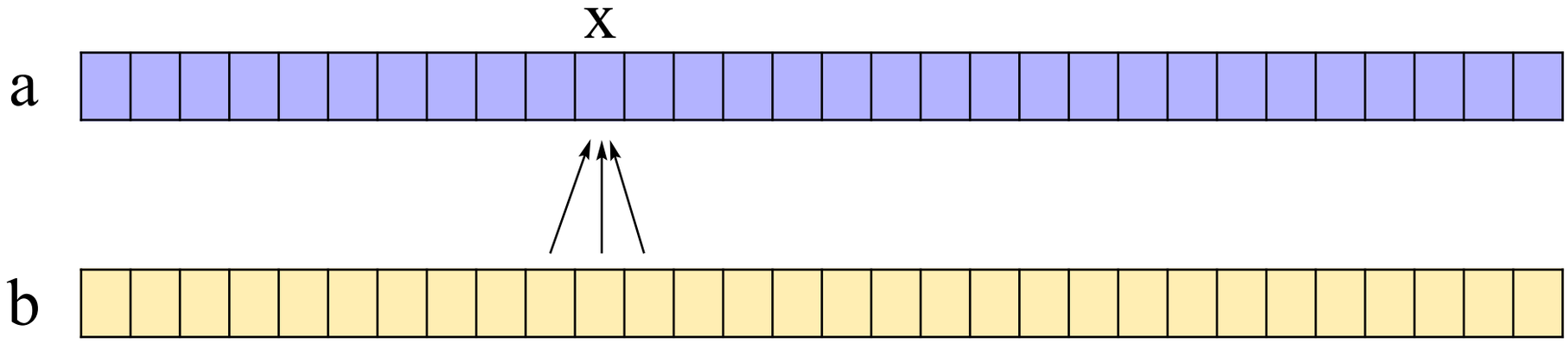


CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap

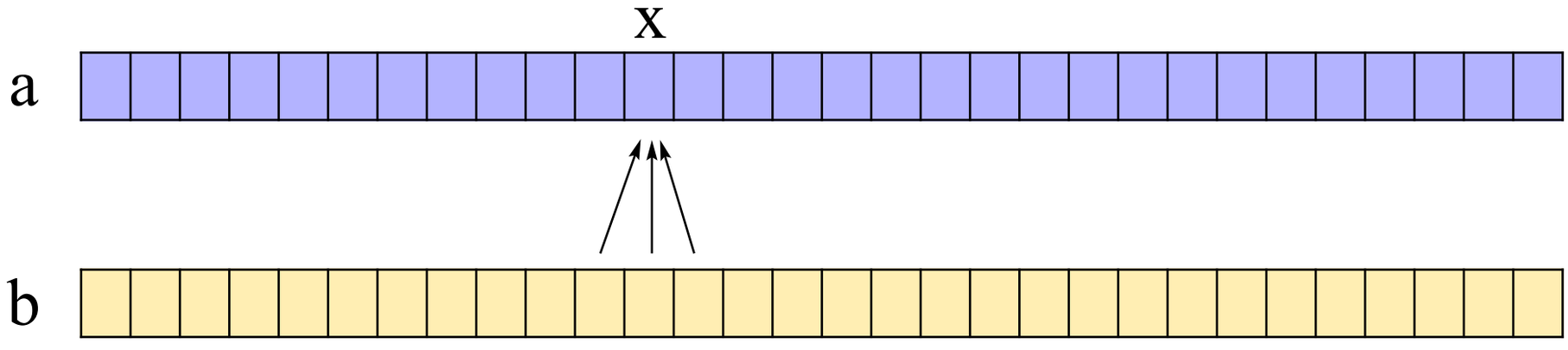


CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap

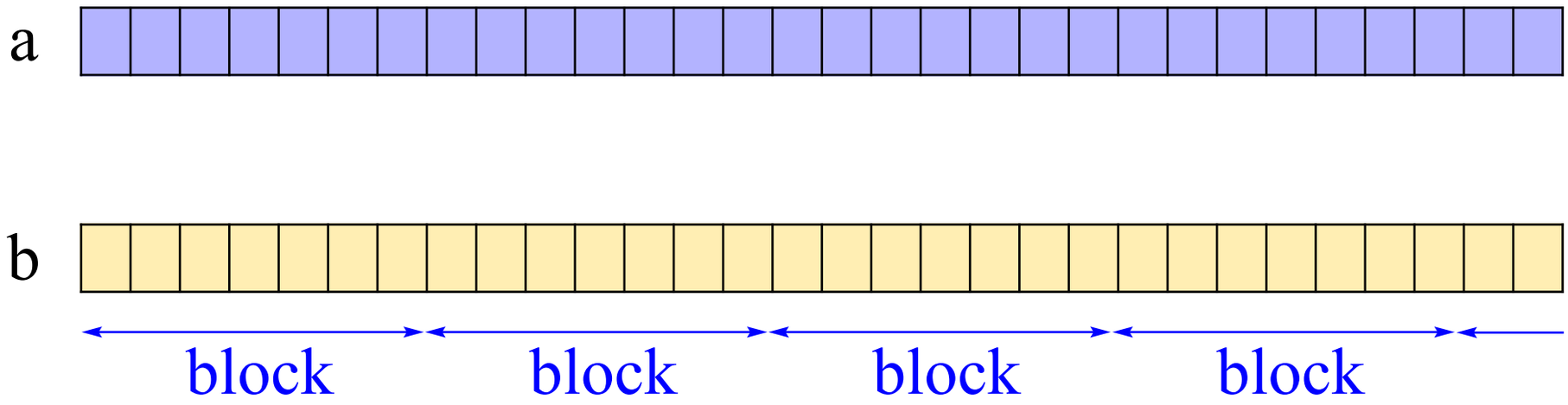


CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap



CUDA Example

```
__global__ void conv( float* in, float* out) {
```

CUDA Example

```
__global__ void conv( float* in, float* out) {

  __shared__ float smem[1024];
```

CUDA Example

```
__global__ void conv( float* in, float* out) {

  __shared__ float smem[1024];

  tid = threadIdx.x + blockDim.x * blockIdx.x;
  smem[threadIdx.x] = in[tid];
  __syncthreads();
```

CUDA Example

```
__global__ void conv( float* in, float* out) {

  __shared__ float smem[1024];

  tid = threadIdx.x + blockDim.x * blockIdx.x;
  smem[threadIdx.x] = in[tid];
  __syncthreads();

  float res = smem[threadIdx.x-1];
  res += 3.0 * smem[threadIdx.x];
  res += smem[threadIdx.x+1];
```

CUDA Example

```
__global__ void conv( float* in, float* out) {

  __shared__ float smem[1024];

  tid = threadIdx.x + blockDim.x * blockIdx.x;
  smem[threadIdx.x] = in[tid];
  __syncthreads();

  float res = smem[threadIdx.x-1];
  res += 3.0 * smem[threadIdx.x];
  res += smem[threadIdx.x+1];

  out[tid] = res;
}
```

CUDA Example

```
__host__ void run() {

  // read input data
  ...

  uint mem_size = signal_size * sizeof(float);
```

CUDA Example

```
__host__ void run() {

  // read input data
  ...

  uint mem_size = signal_size * sizeof(float);

  float* d_in, d_out;
  cudaMalloc( (void**) &d_in, mem_size);
  cudaMalloc( (void**) &d_out, mem_size);
  cudaMemcpy( d_in, signal, mem_size,
              cudaMemcpyHostToDevice);
```

CUDA Example

```
// assume signal_size > 512
dim3 threads, blocks;
threads = 512;
blocks = (signal_size / 512) + 1;
```

CUDA Example

```
// assume signal_size > 512
dim3 threads, blocks;
threads = 512;
blocks = (signal_size / 512) + 1;

// run kernel
conv<<<blocks,threads>>>( d_in, d_out);
```

CUDA Example

```
    // assume signal_size > 512
    dim3 threads, blocks;
    threads = 512;
    blocks = (signal_size / 512) + 1;

    // run kernel
    conv<<<blocks,threads>>>( d_in, d_out);
    cudaThreadSynchronize();

    // copy result back to the host array result
    cudaMemcpy( result, d_out, mem_size,
                cudaMemcpyDeviceToHost);

}
```

Final Remarks

- Data parallel co-processor: task parallel array of data parallel multiprocessor.
- Efficient programs require to keep hardware architecture in mind.
  - Execution model and organization of threads.
  - Memory hierarchy and latencies.