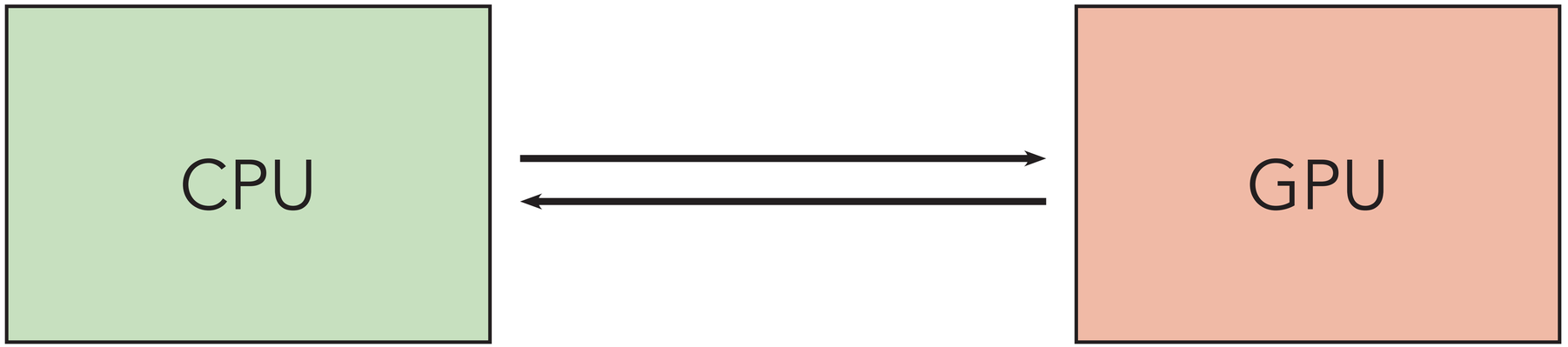
GPU Programming

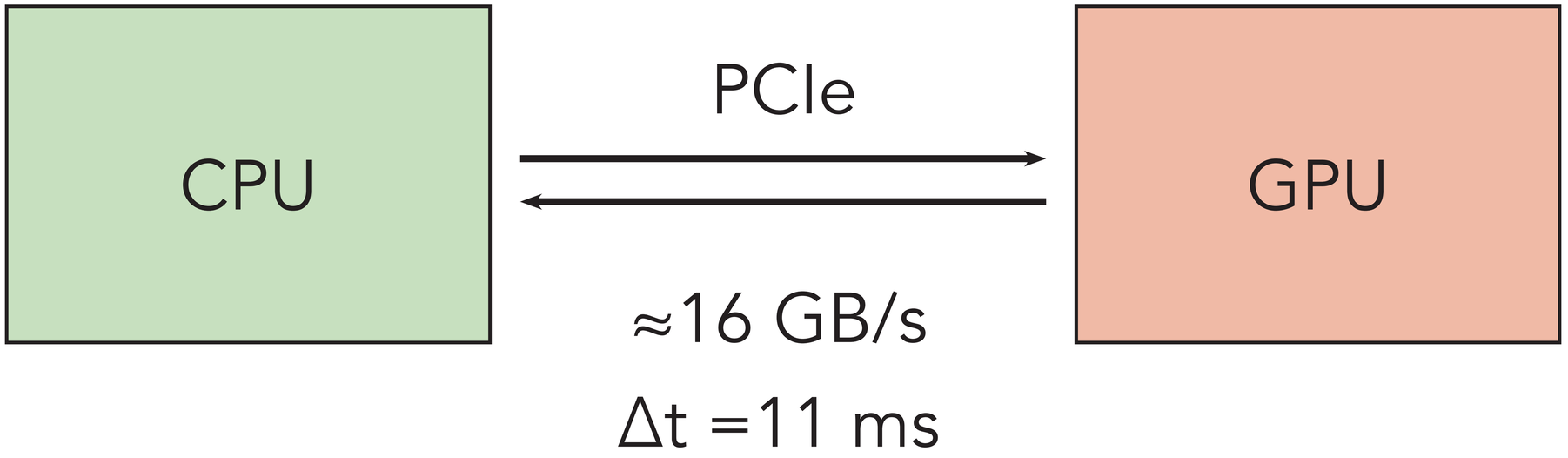
Data-Parallel Co-Processors

Christian Lessig

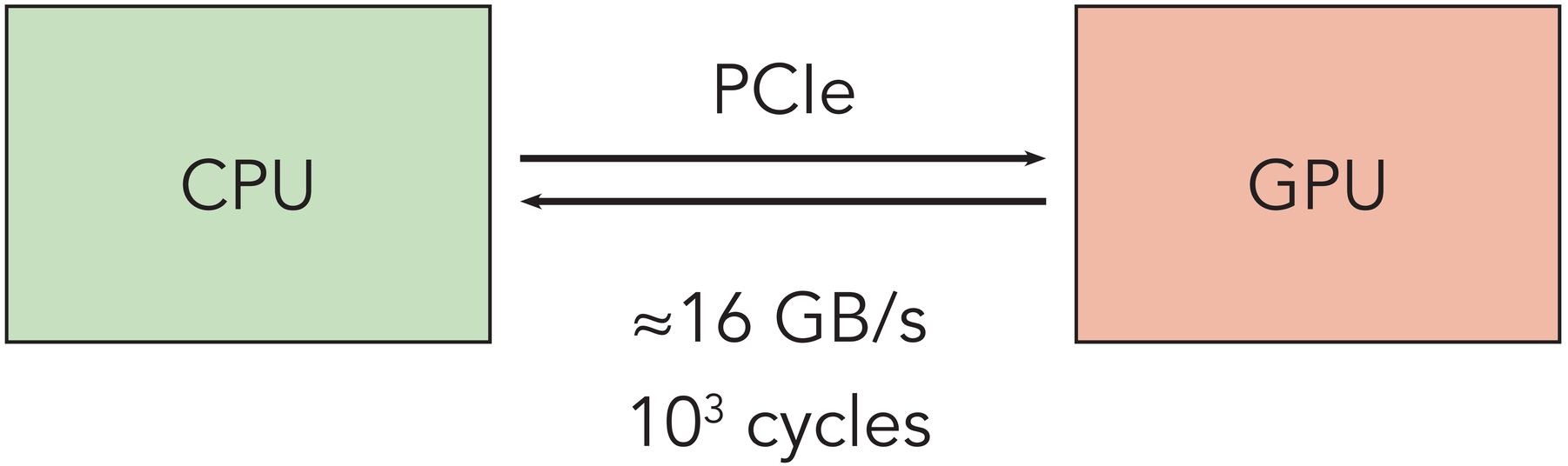
Data parallel co-processors



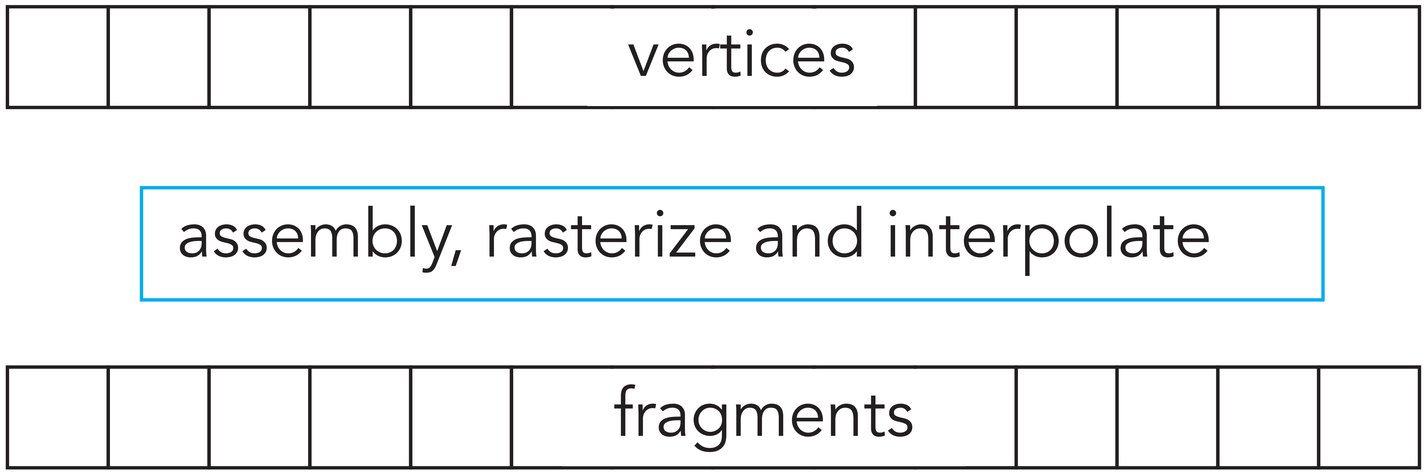
Data parallel co-processors



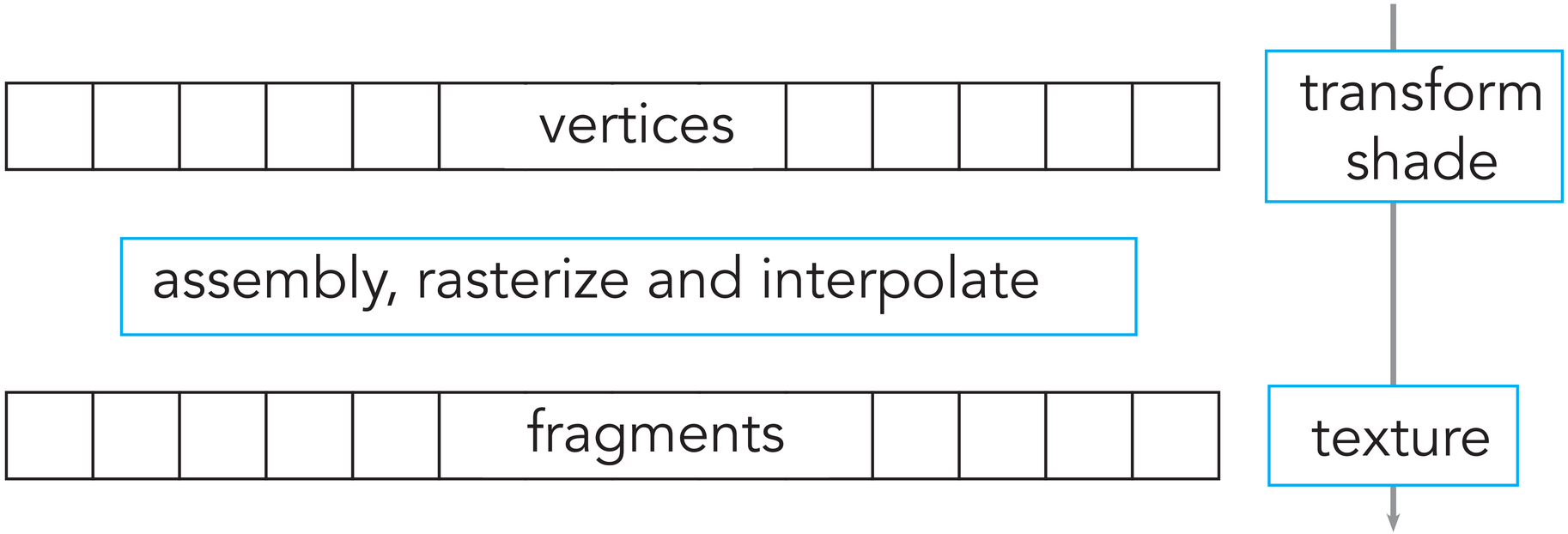
Data parallel co-processors



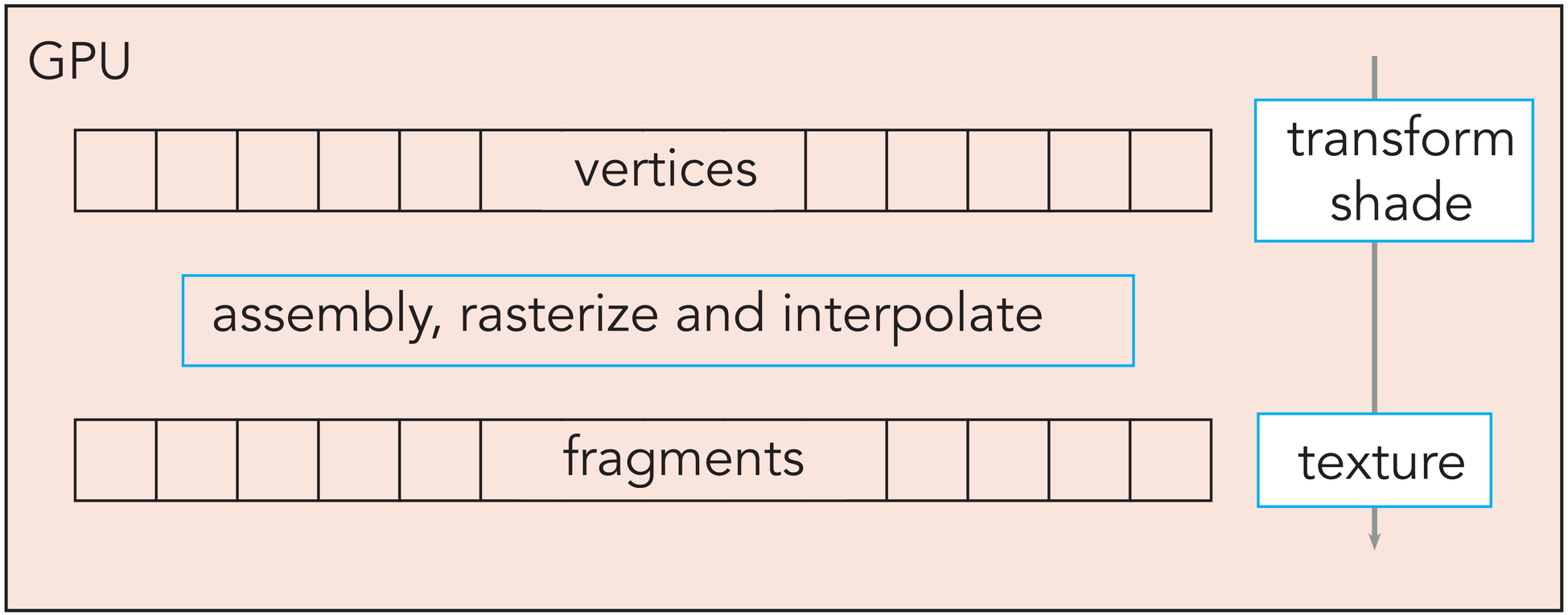
Rasterization pipeline



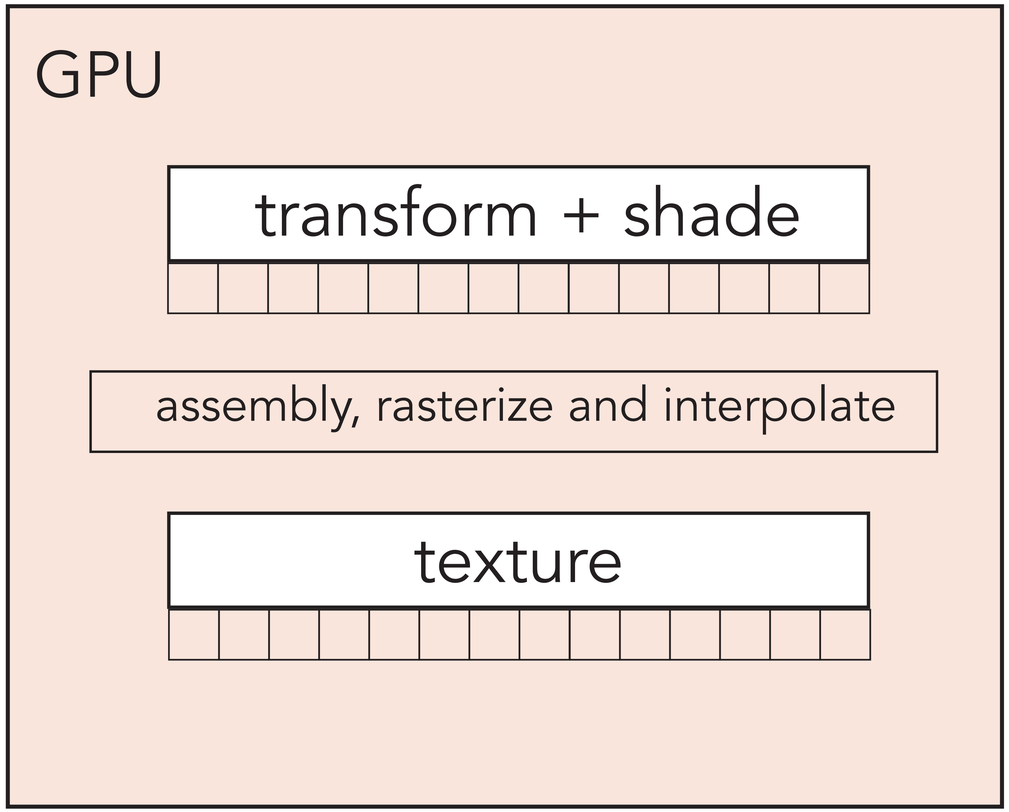
Rasterization pipeline



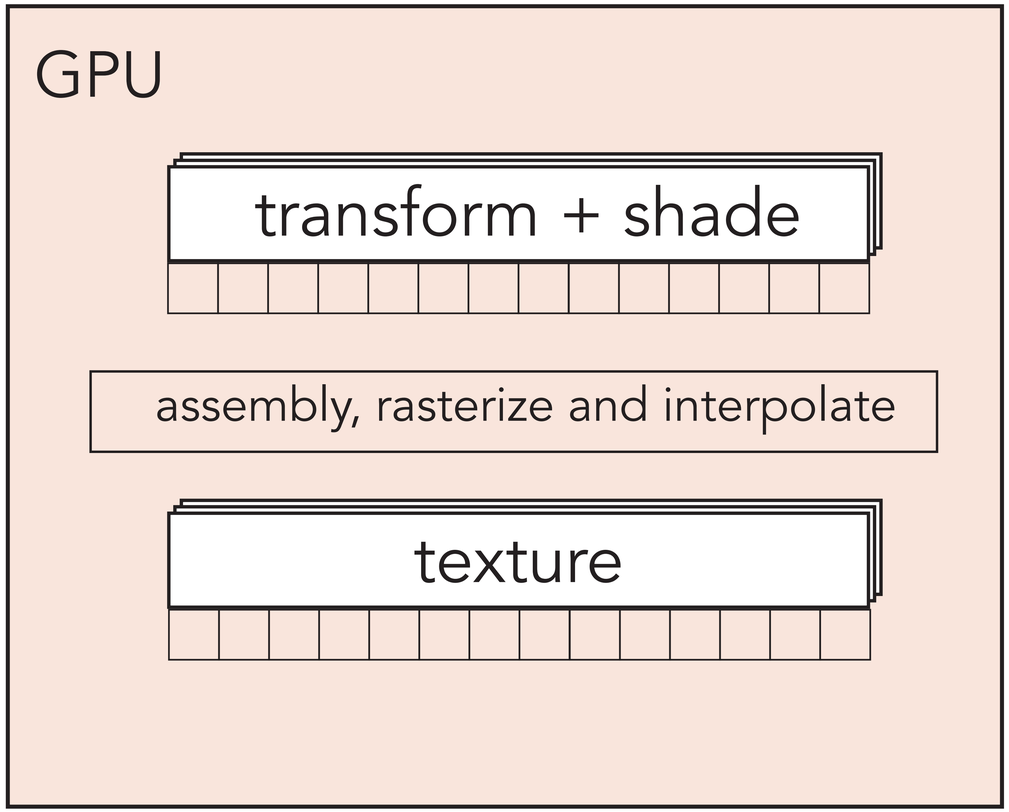
Graphics co-processors



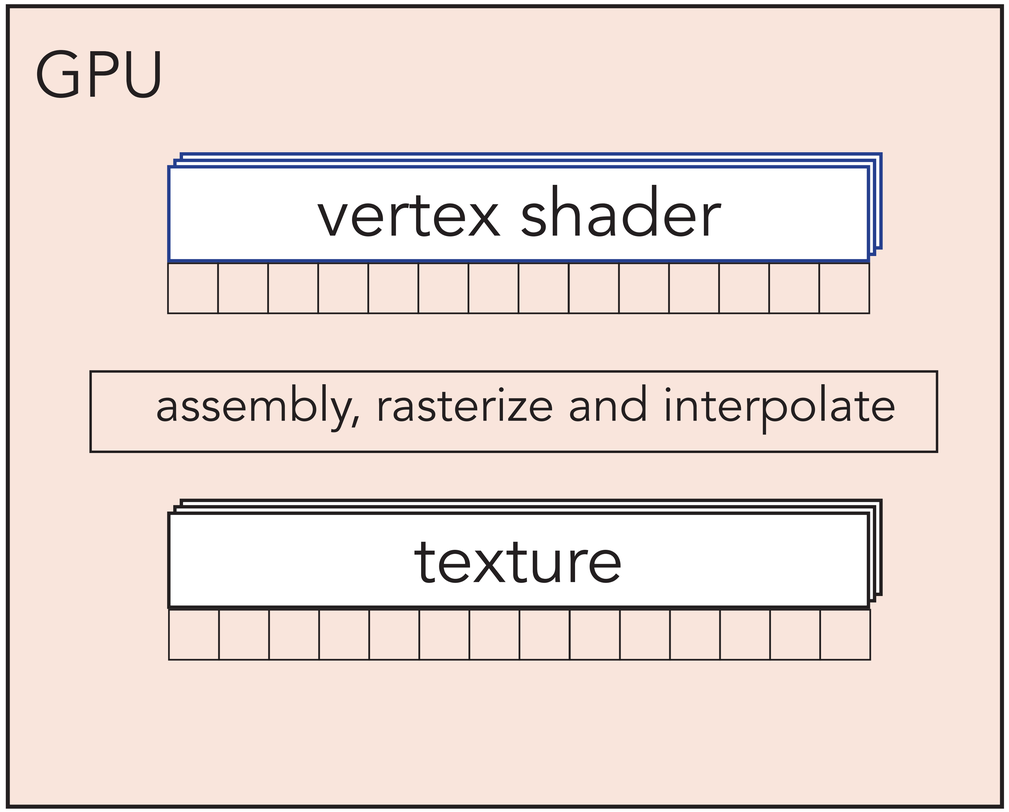
Graphics co-processors



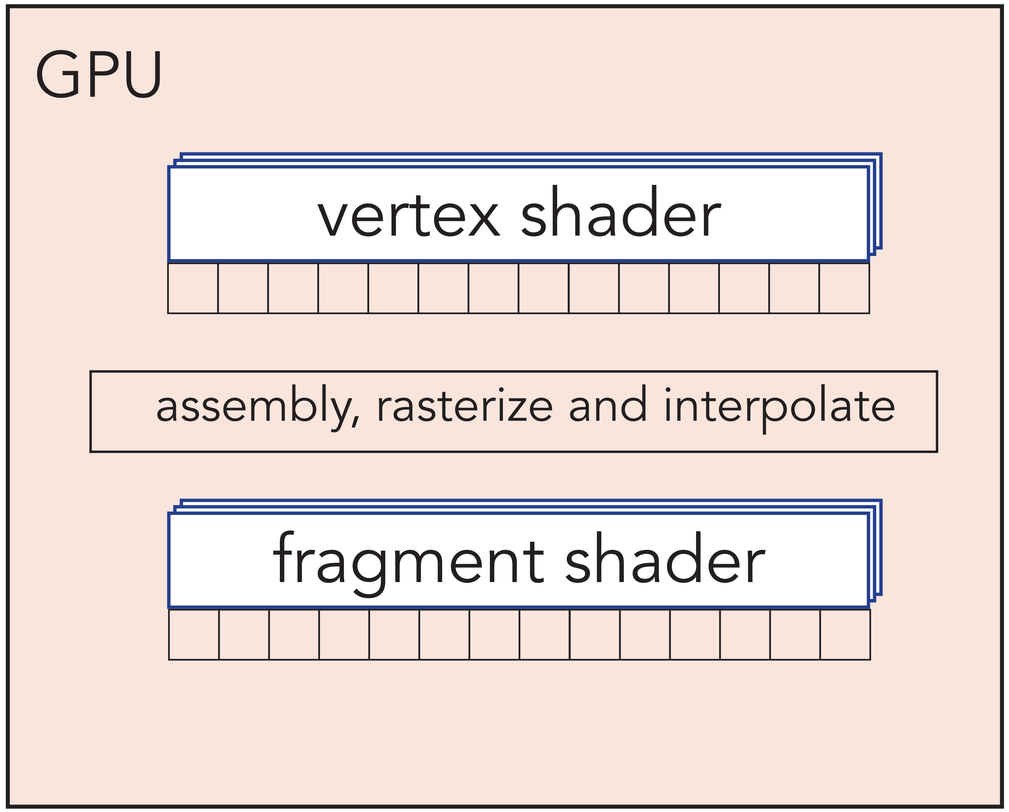
Graphics co-processors



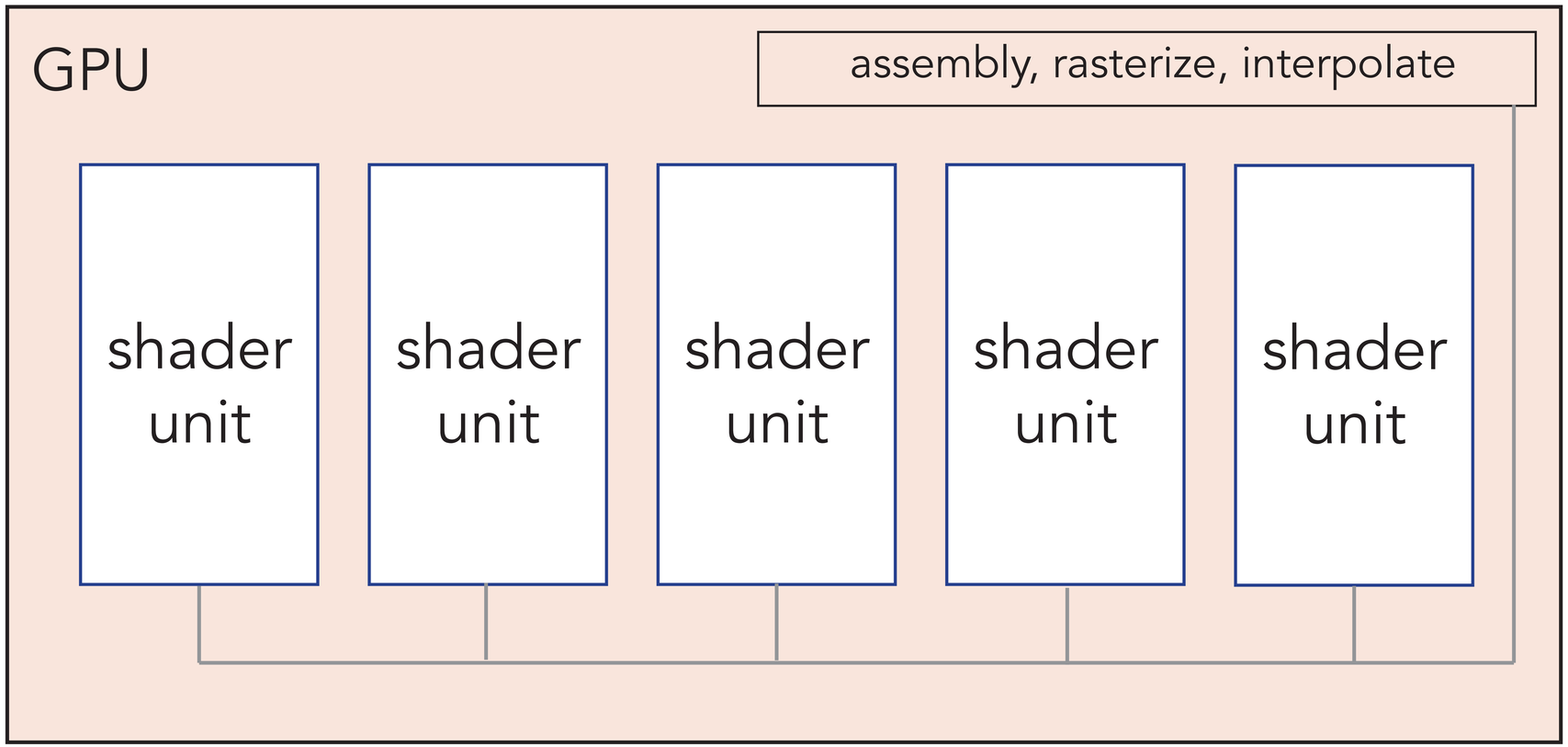
Graphics co-processors



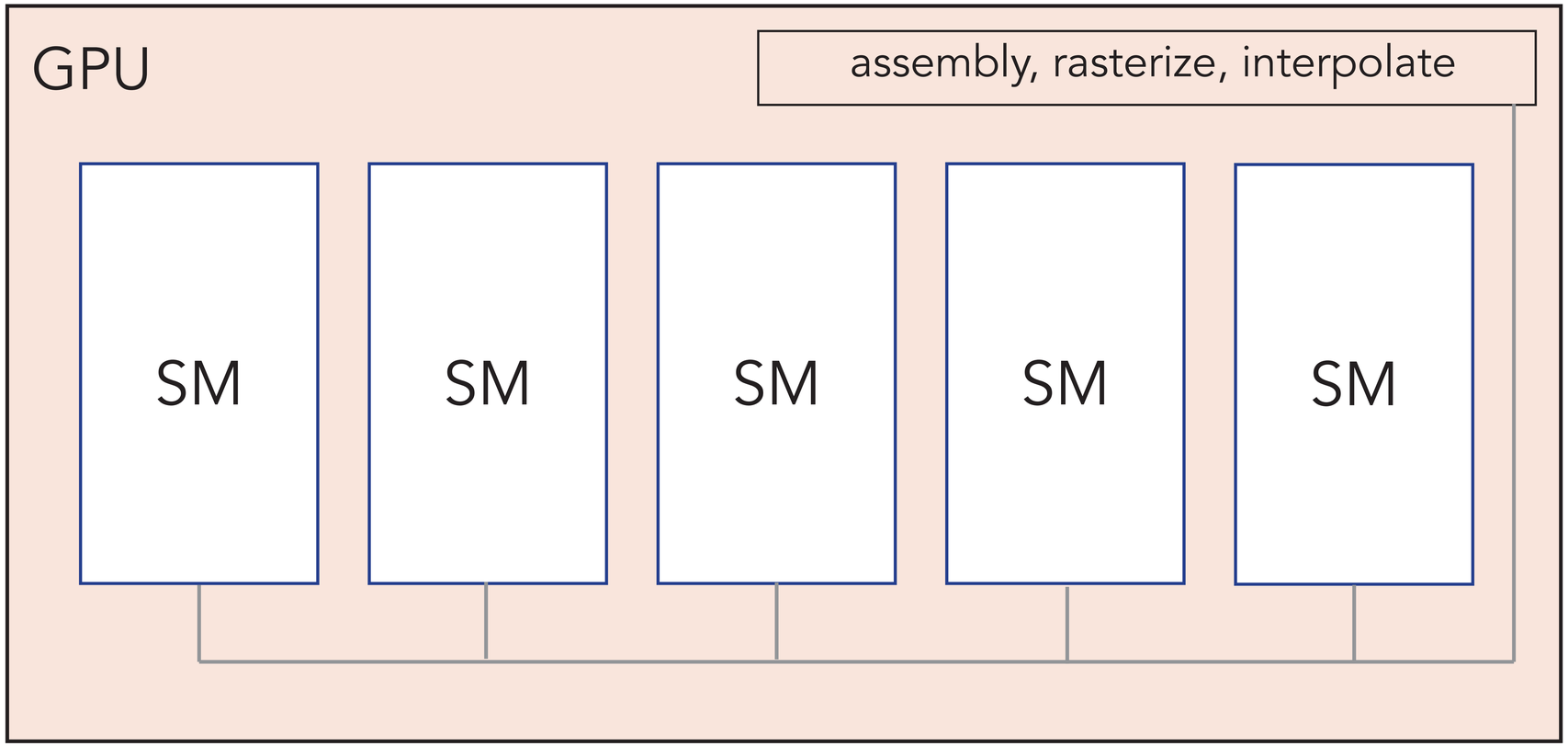
Graphics co-processors



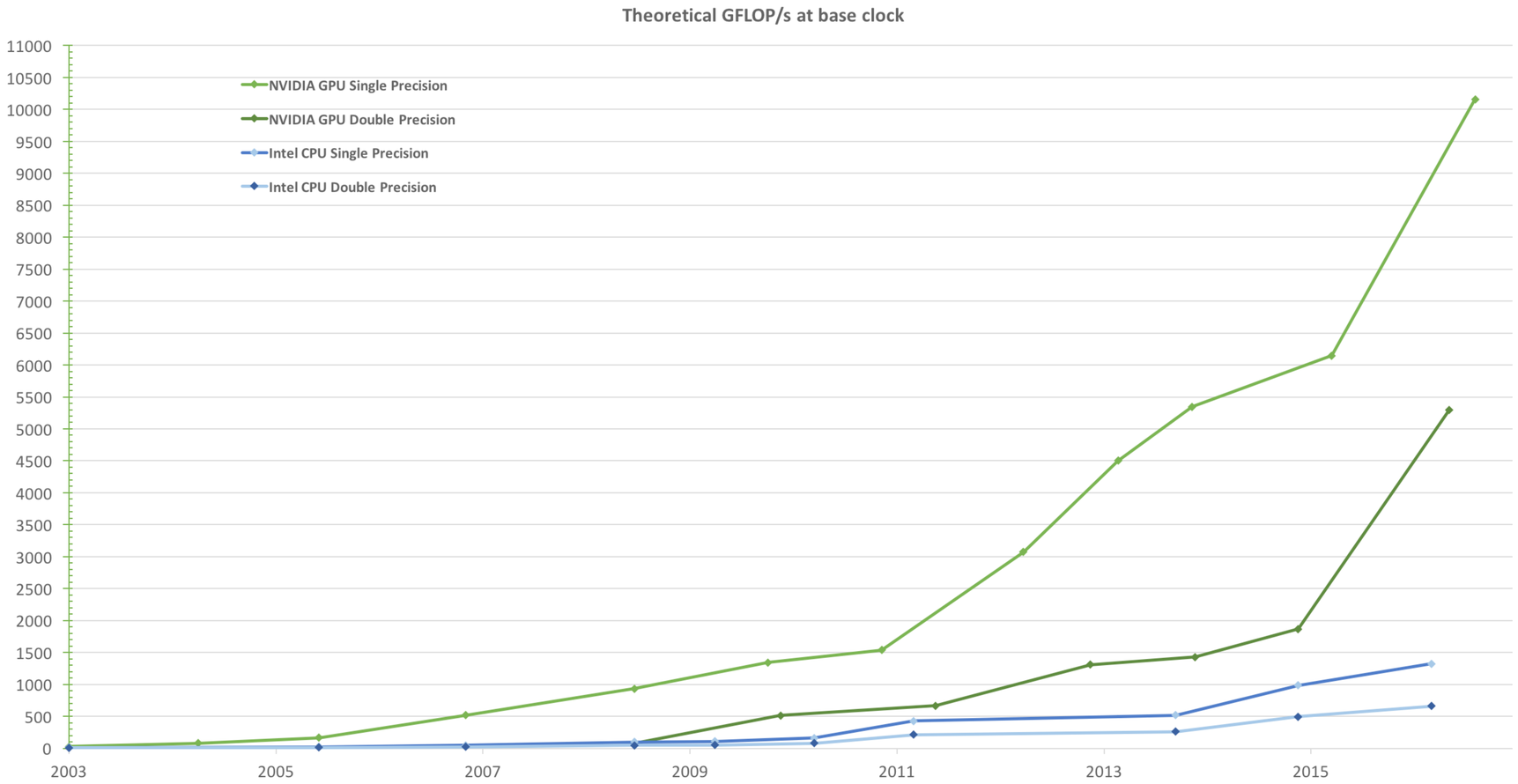
Graphics co-processors



Data parallel co-processors



Data parallel co-processors



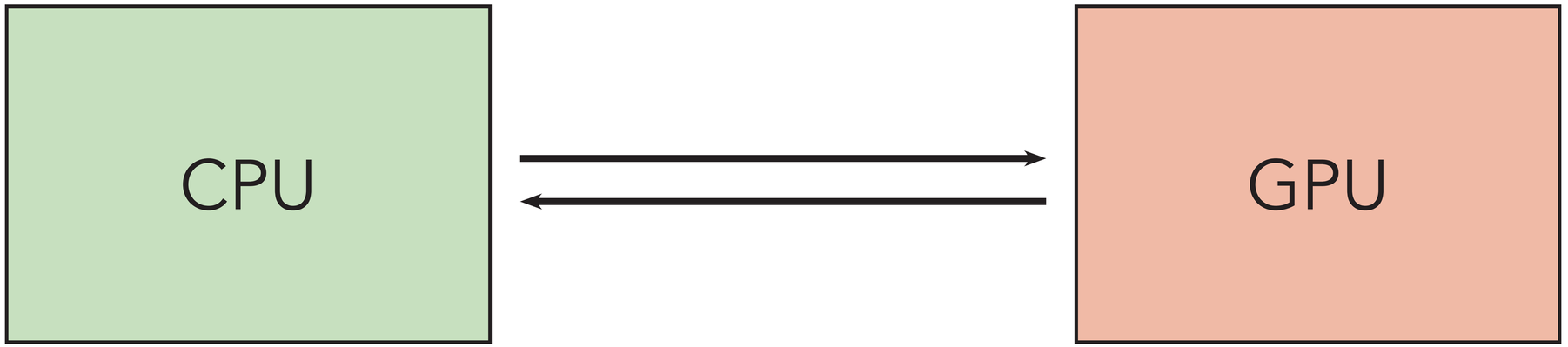
from CUDA C Programming Guide

Data parallel co-processors

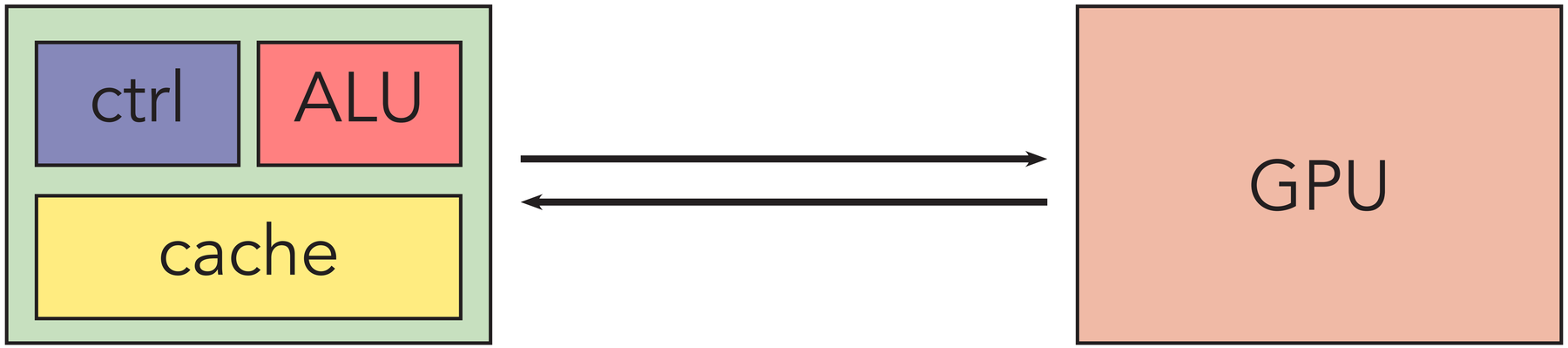
"Combined with a pixel shader workload that is usually compute limited, these characteristics have guided GPUs along a different evolutionary path than CPUs. In particular, whereas the CPU die area is dominated by cache memories, GPUs are dominated by floating-point datapath and fixed-function logic. GPU memory interfaces emphasize bandwidth over latency (as latency can be readily hidden by massively parallel execution); indeed, bandwidth is typically many times higher than that for a CPU, exceeding 100 GB/s in more recent designs."

D. Kirk and W.-m. Hwu

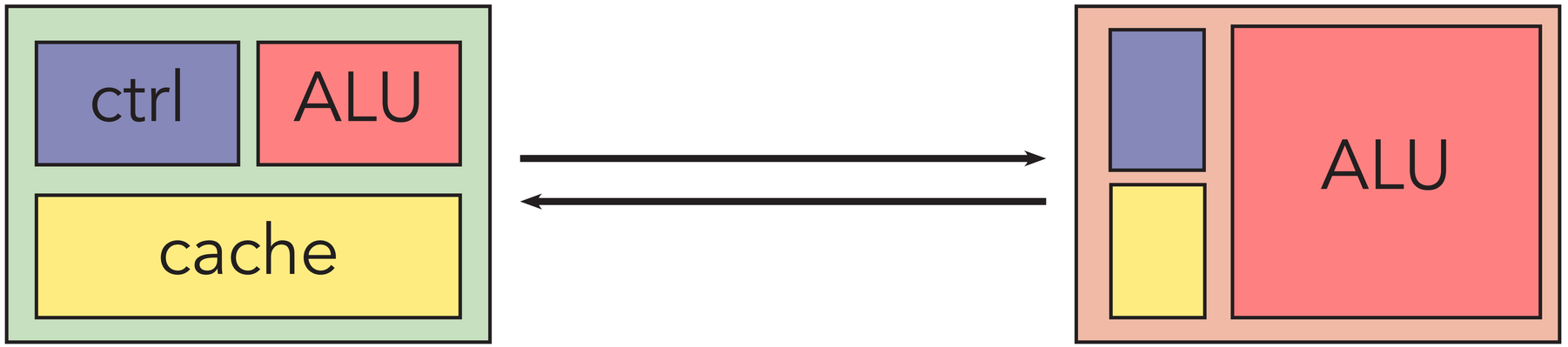
Data parallel co-processors



Data parallel co-processors



Data parallel co-processors



Data parallel co-processors

	CPU	GPU
clock		
ALU latency		
mem latency		
bandwidth		

CUDA C Programming Guide, Ch. 5, compute capabilities 3.x

Data parallel co-processors

	CPU	GPU
clock	3 GHz	1 GHz
ALU latency		
mem latency		
bandwidth		

CUDA C Programming Guide, Ch. 5, compute capabilities 3.x

Data parallel co-processors

	CPU	GPU
clock	3 GHz	1 GHz
ALU latency	3 cycles	11 cycles
mem latency		
bandwidth		

CUDA C Programming Guide, Ch. 5, compute capabilities 3.x

Data parallel co-processors

	CPU	GPU
clock	3 GHz	1 GHz
ALU latency	3 cycles	11 cycles
mem latency	30 cycles	300 cycles
bandwidth		

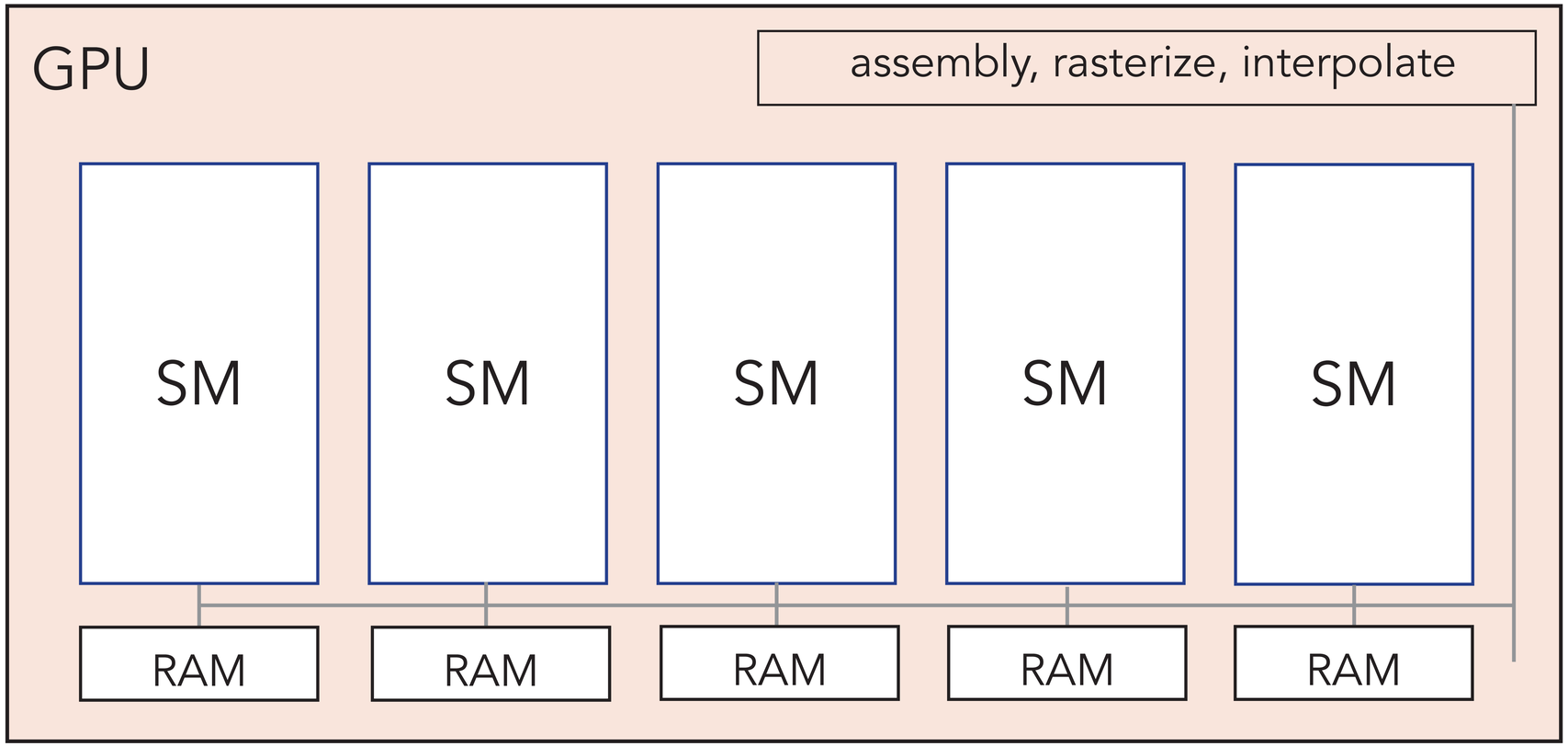
CUDA C Programming Guide, Ch. 5, compute capabilities 3.x

Data parallel co-processors

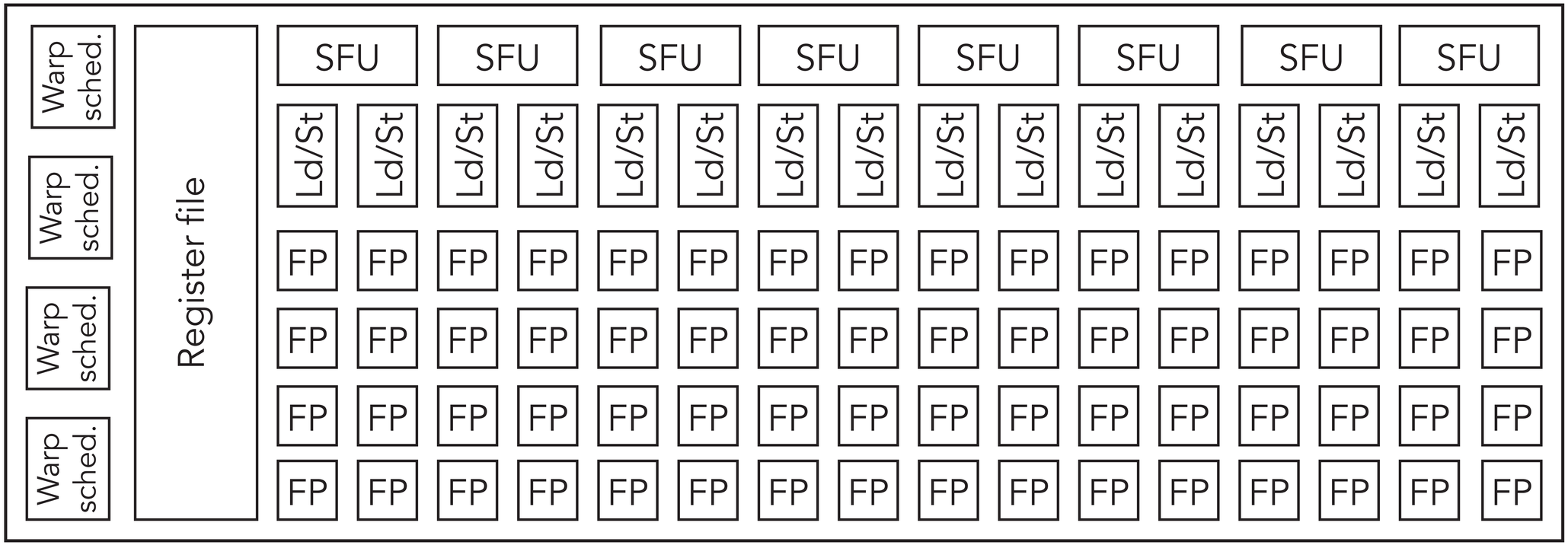
	CPU	GPU
clock	3 GHz	1 GHz
ALU latency	3 cycles	11 cycles
mem latency	30 cycles	300 cycles
bandwidth	80 GB/s	800 GB/s

CUDA C Programming Guide, Ch. 5, compute capabilities 3.x

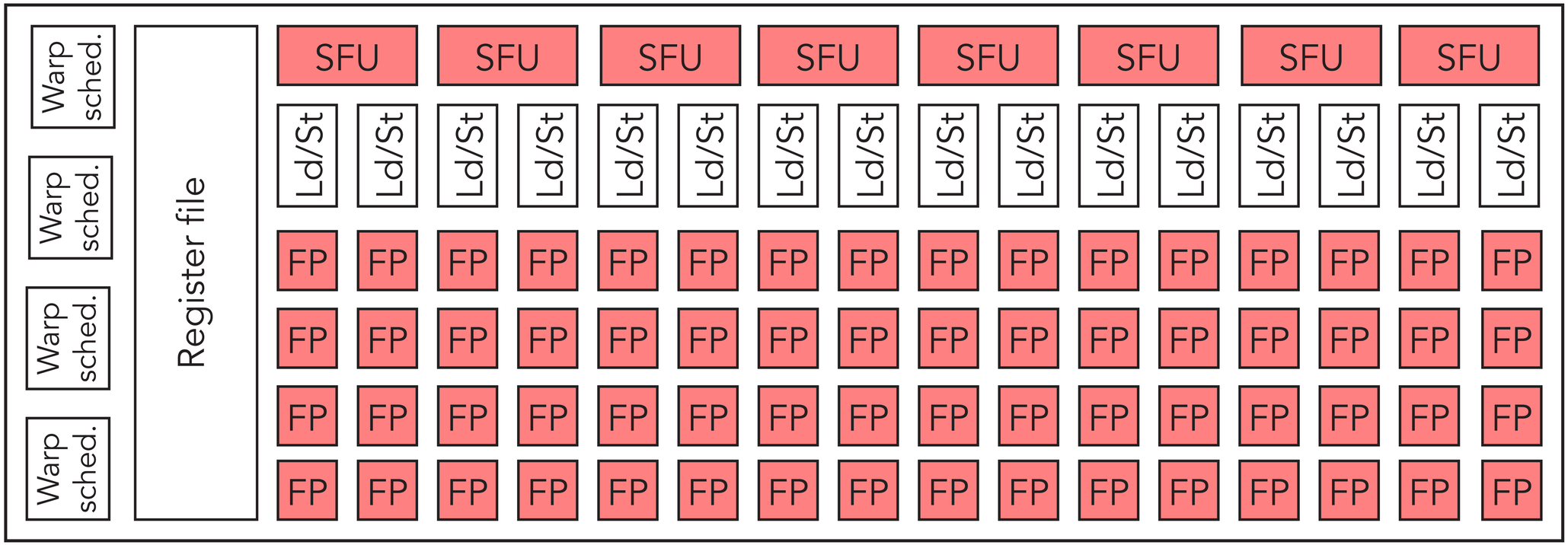
Data parallel co-processors



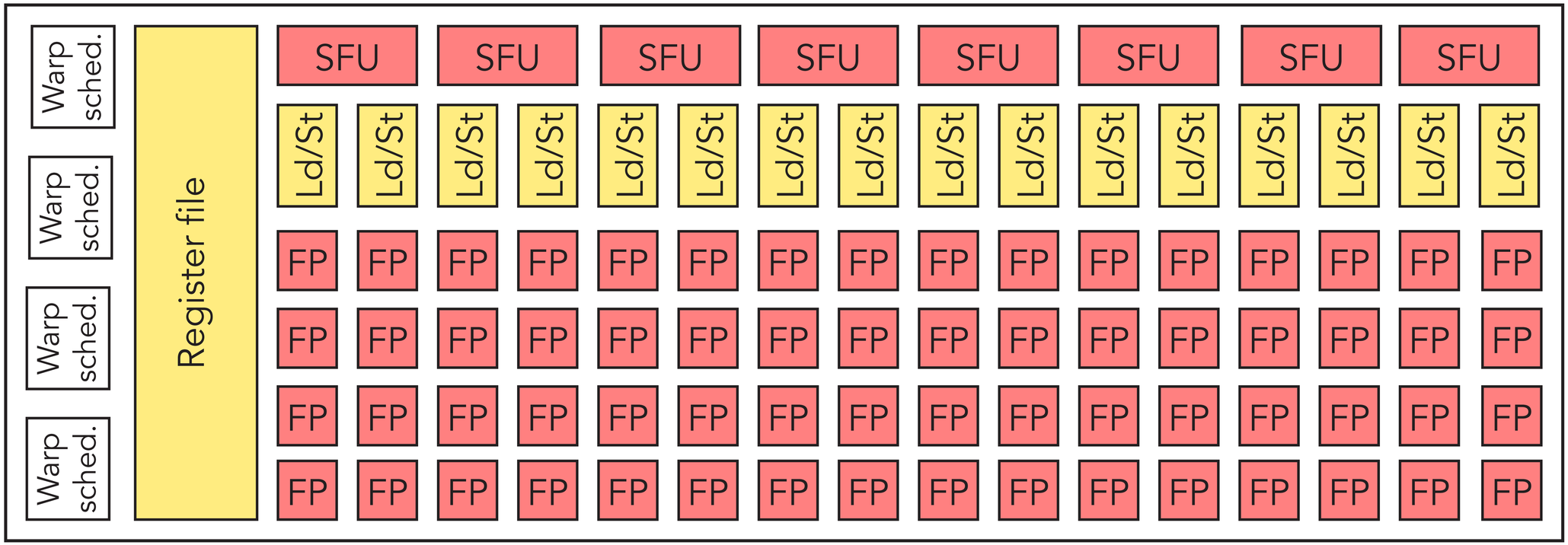
Data parallel co-processors



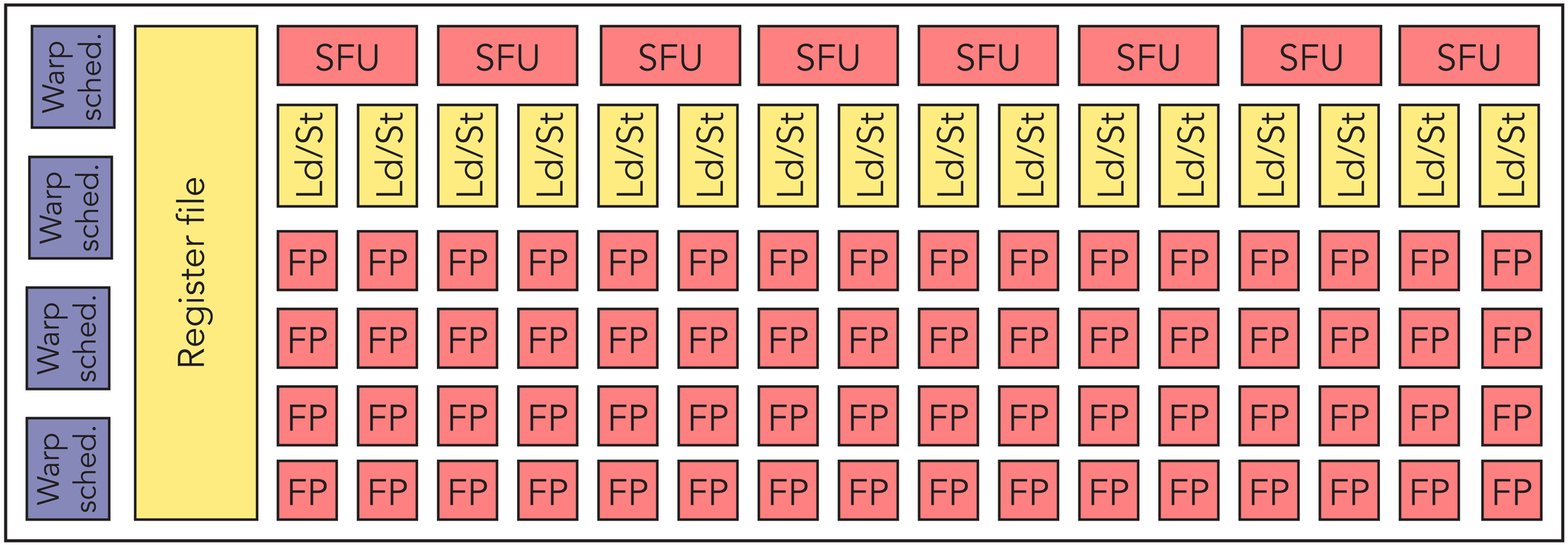
Data parallel co-processors



Data parallel co-processors



Data parallel co-processors



Cuda

- Developed around 2005
- Originates in Brook project at Stanford
- Based on streaming paradigm
- C/C++ API++
  - separate compiler
  - runtime API: high-level API (`cuda*()`)
  - driver API: low-level API (`cu*()`)

OpenCL

- Open standard for parallel programming
- For hardware consisting of CPUs, GPUs, FPGAs, ...
- Aims at abstraction from actual hardware configuration on which code is executed
  - But performance depends in general significantly on hardware configuration
- Implementations by Intel, AMD, IBM, (Nvidia)

OpenCL versus Cuda

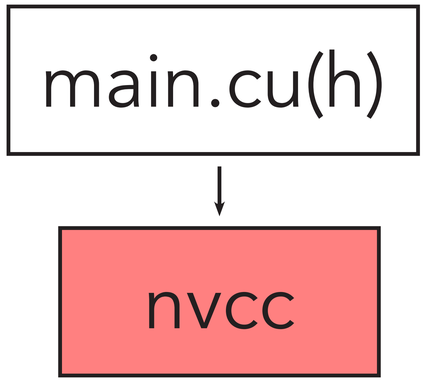
- For CPU-GPU configuration code can be translated forth and back between APIs
  - Same underlying programming model
- CUDA syntax generally considered somewhat simpler and cleaner
- OpenCL only lukewarm support from Nvidia
  - Better performance using Cuda

Cuda

main.cu(h)

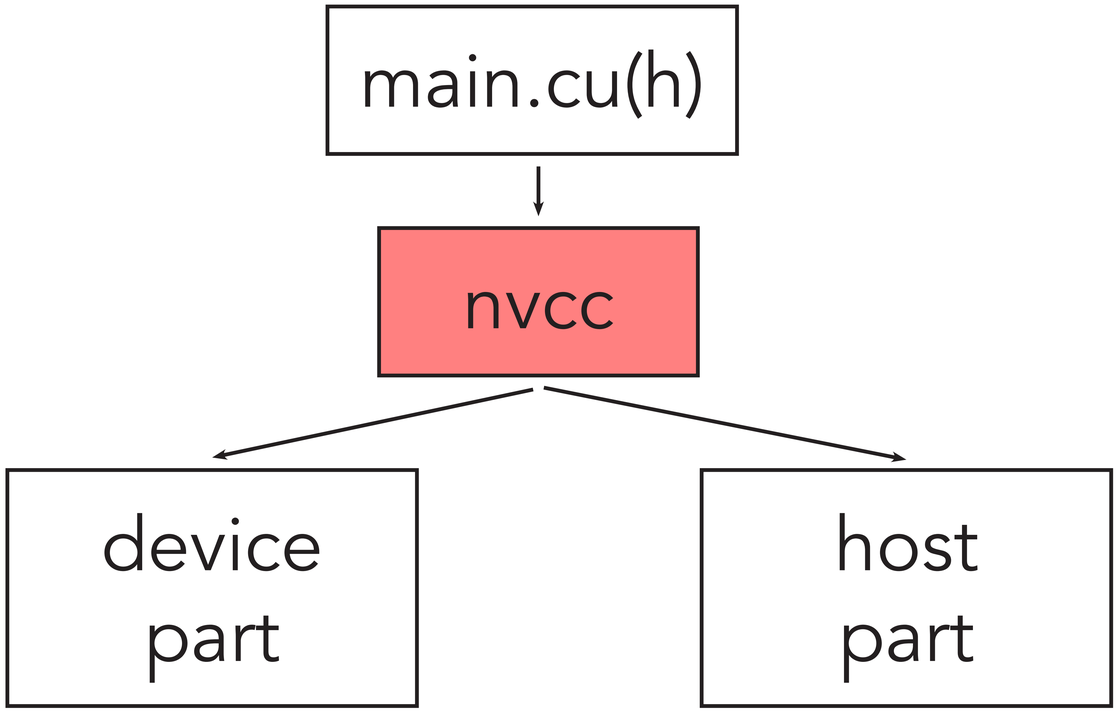
Details: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

Cuda



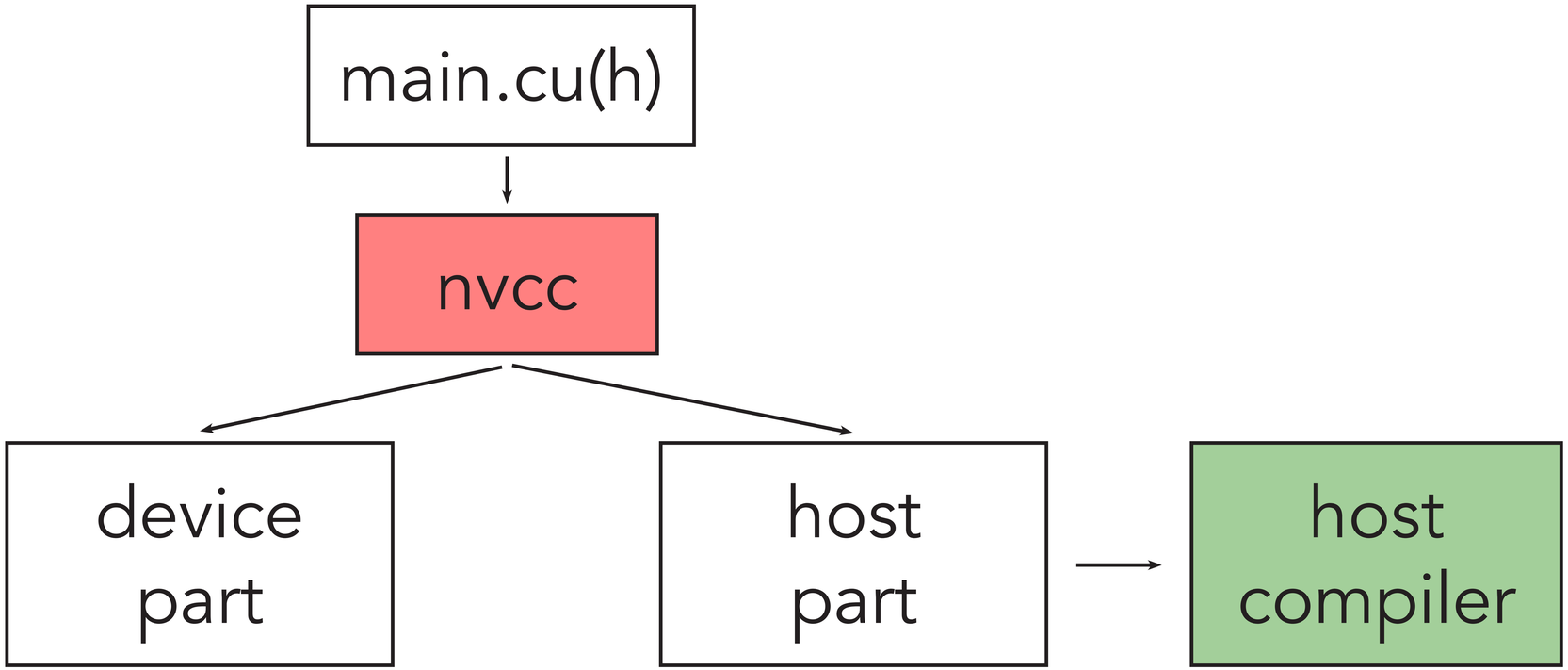
Details: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

Cuda



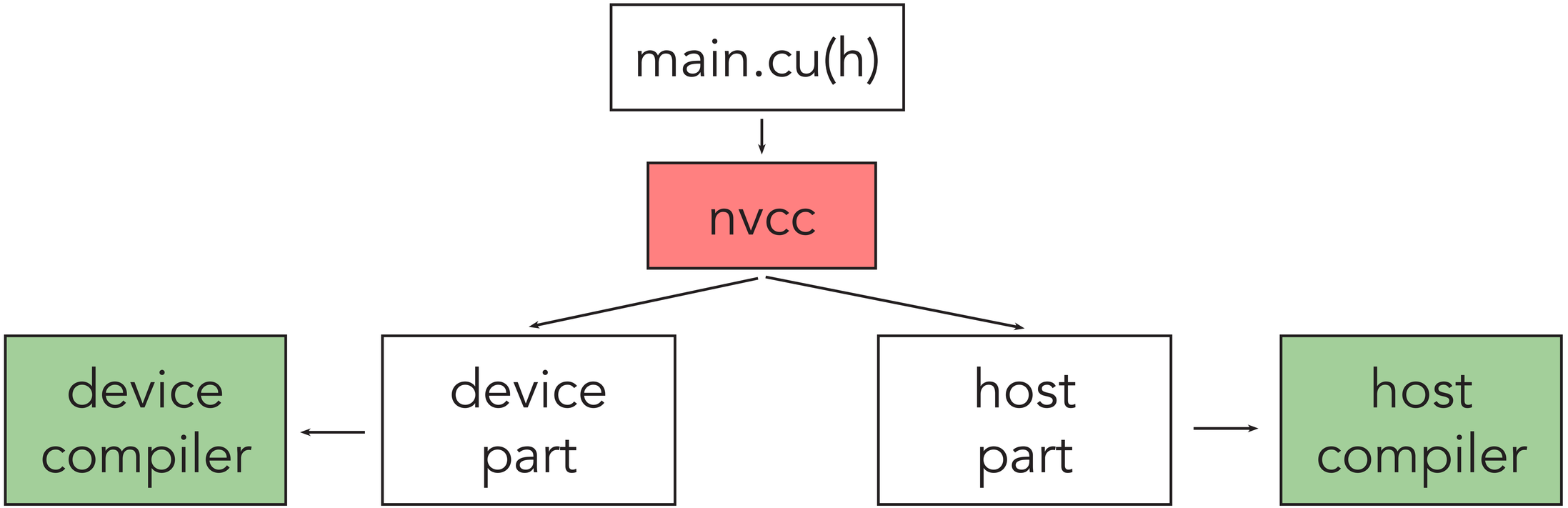
Details: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

Cuda



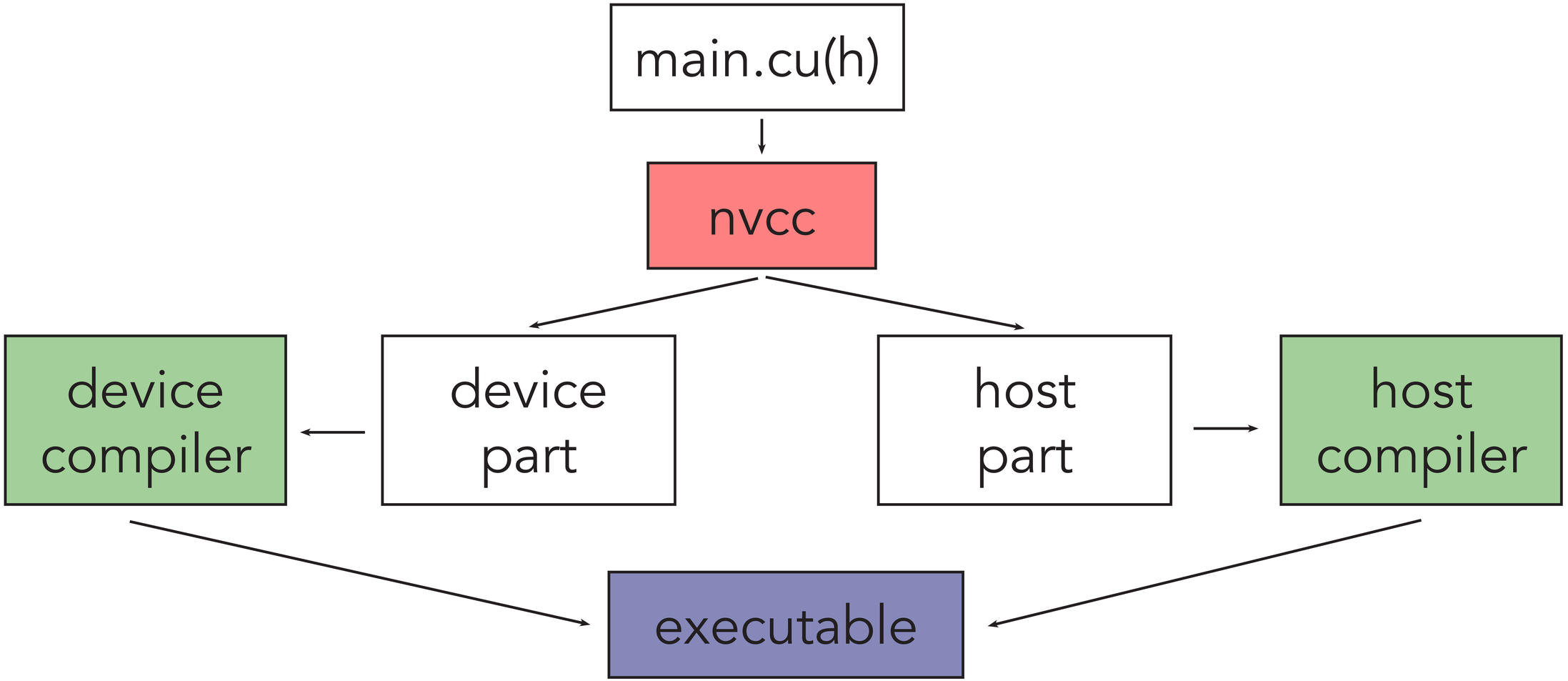
Details: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

Cuda



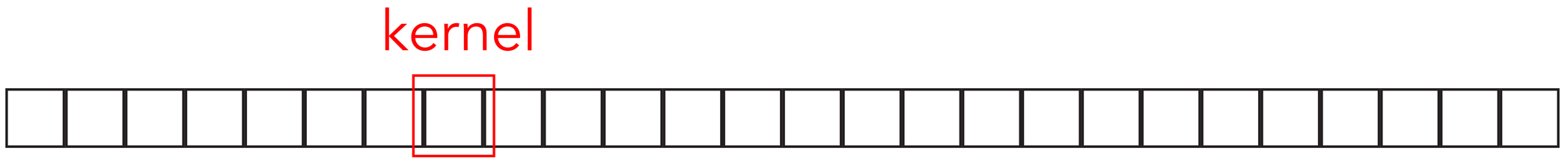
Details: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

Cuda

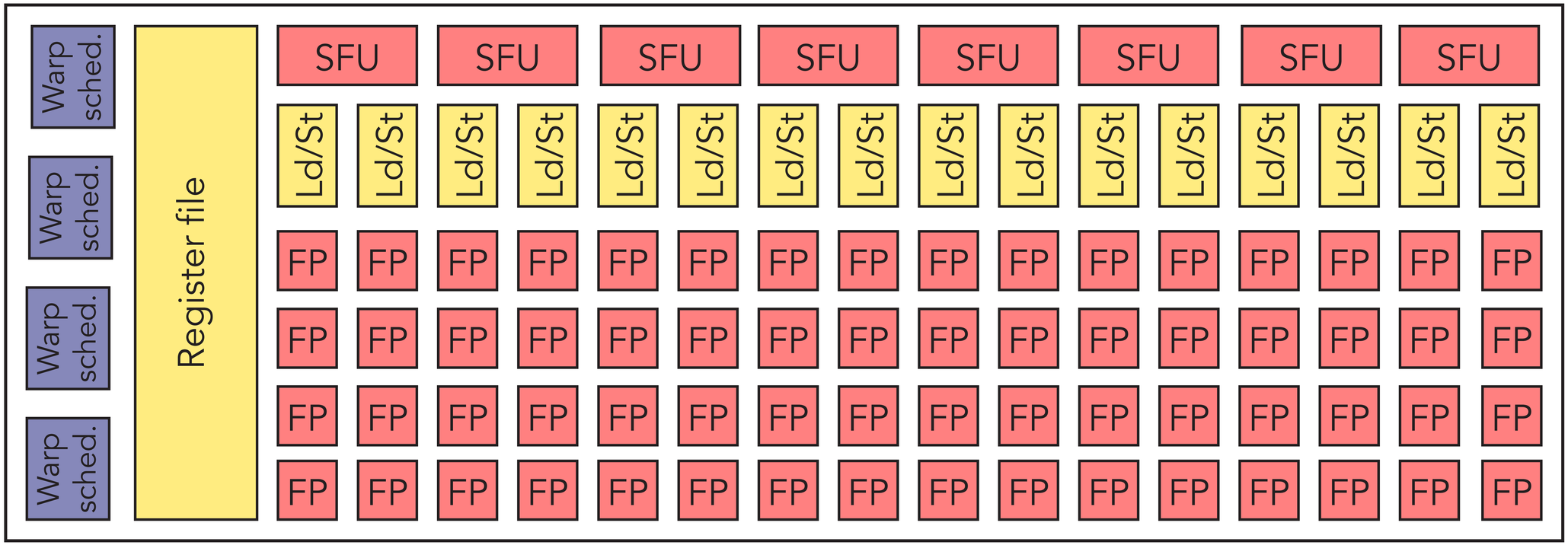


Details: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

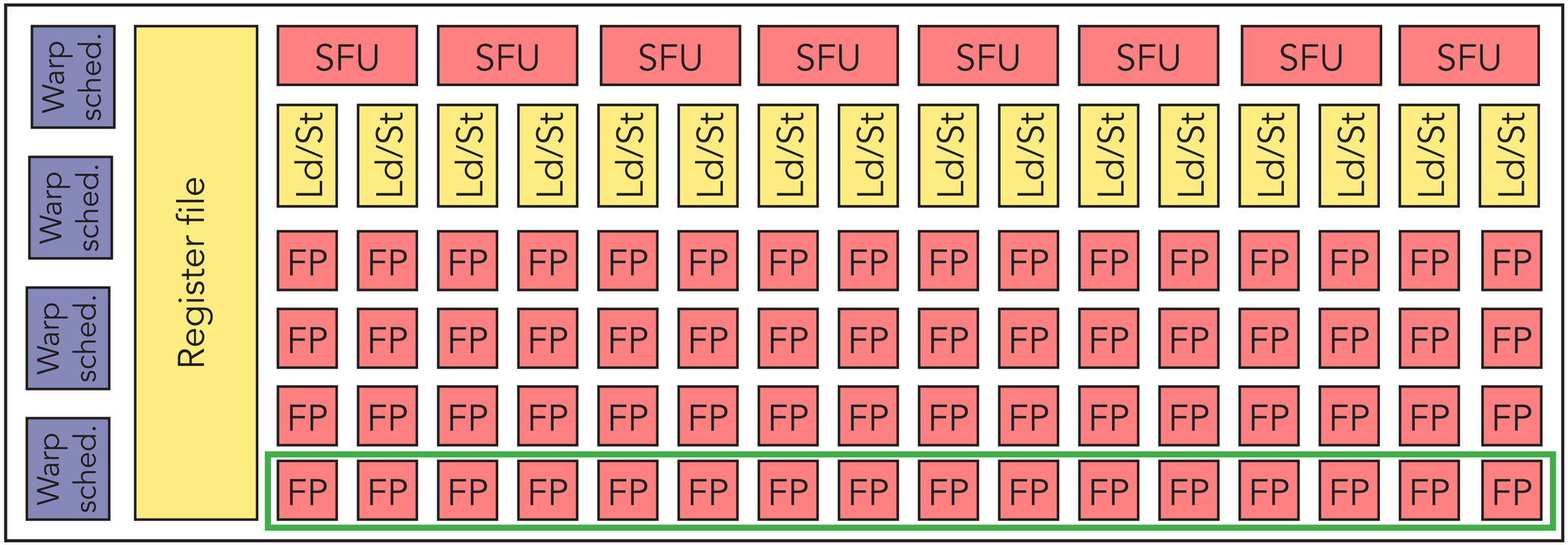
Cuda: device abstraction



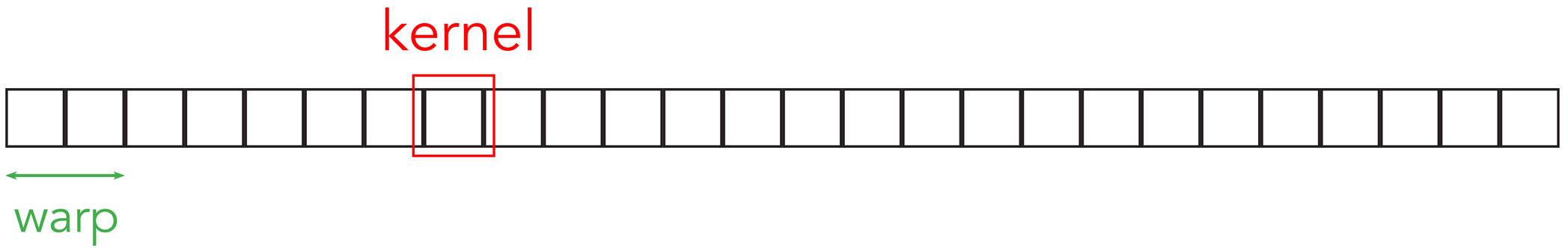
Cuda: device abstraction



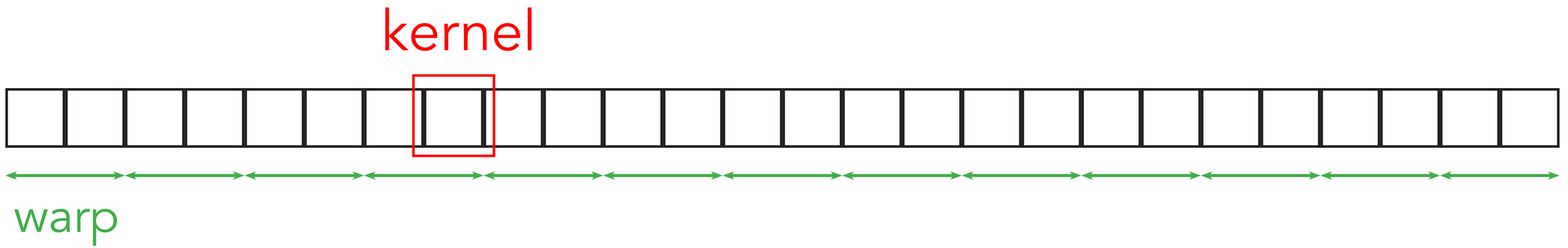
Cuda: device abstraction



Cuda: device abstraction



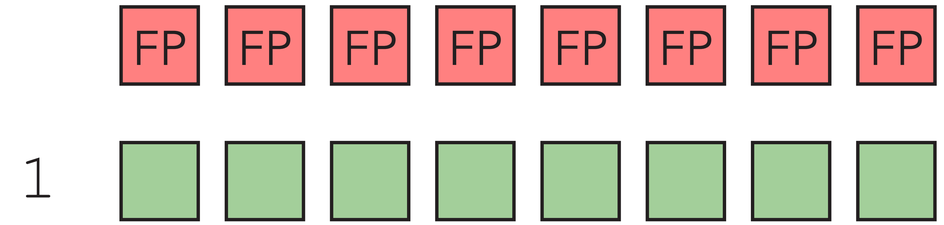
Cuda: device abstraction



Cuda: device abstraction


```
data[tid] += 4;
```

Cuda: device abstraction

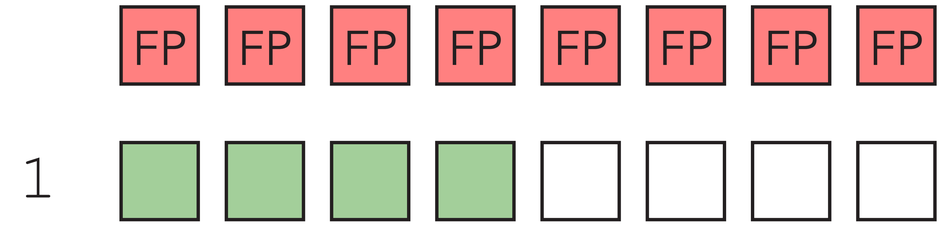



```
data[tid] += 4;
```

Cuda: device abstraction

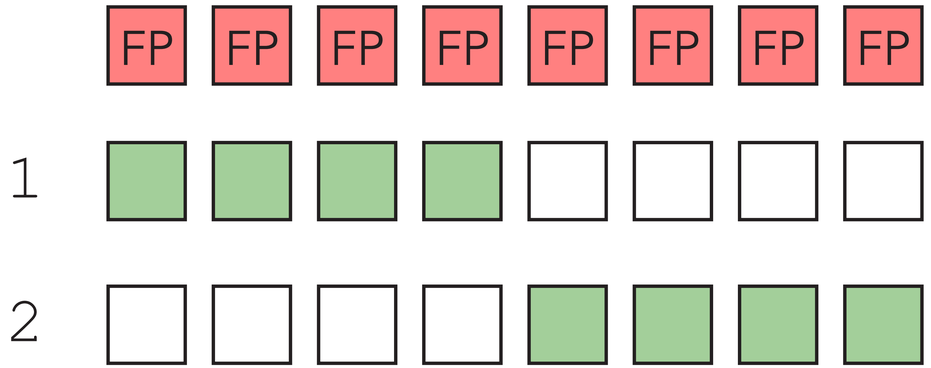

```
if( tid < 4) {
   data[tid] += 4;
}
else {
   data[tid] -= 4;
}
```

Cuda: device abstraction



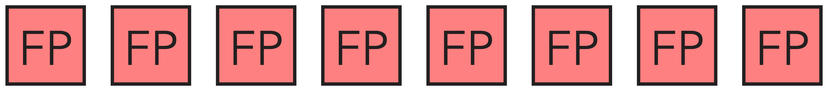

```
if( tid < 4) {
   data[tid] += 4;
}
else {
   data[tid] -= 4;
}
```

Cuda: device abstraction



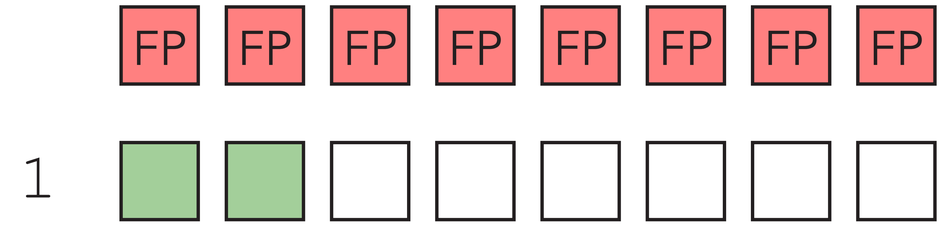

```
if( tid < 4) {
   data[tid] += 4;
}
else {
   data[tid] -= 4;
}
```

Cuda: device abstraction



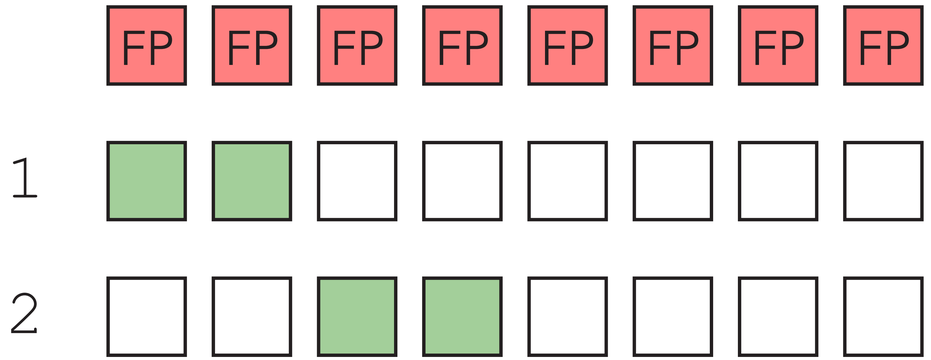

```
if( tid < 2)
  data[tid] += 4;
}
else if( tid < 4) {
  data[tid] -= 4;
}
else if( tid < 6) {
  data[tid] *= 4;
}
else {
  data[tid] /= 4;}
```

Cuda: device abstraction



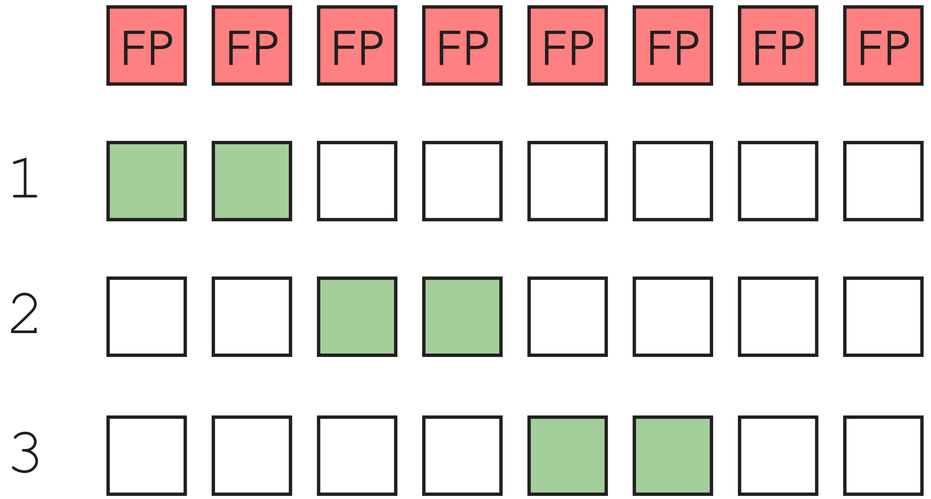

```
if( tid < 2)
  data[tid] += 4;
}
else if( tid < 4) {
  data[tid] -= 4;
}
else if( tid < 6) {
  data[tid] *= 4;
}
else {
  data[tid] /= 4;}
```

Cuda: device abstraction



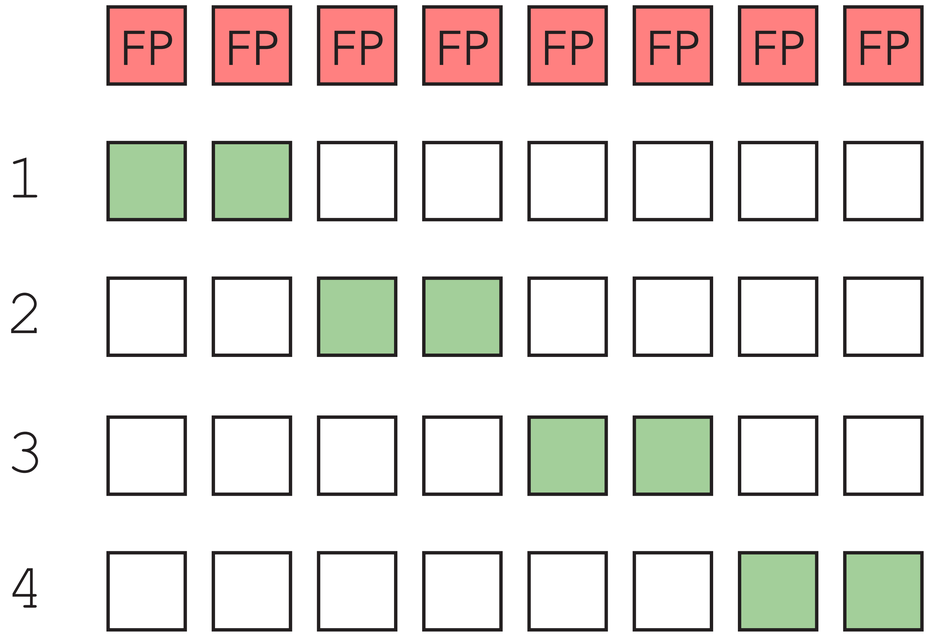

```
if( tid < 2)
  data[tid] += 4;
}
else if( tid < 4) {
  data[tid] -= 4;
}
else if( tid < 6) {
  data[tid] *= 4;
}
else {
  data[tid] /= 4;}
```

Cuda: device abstraction



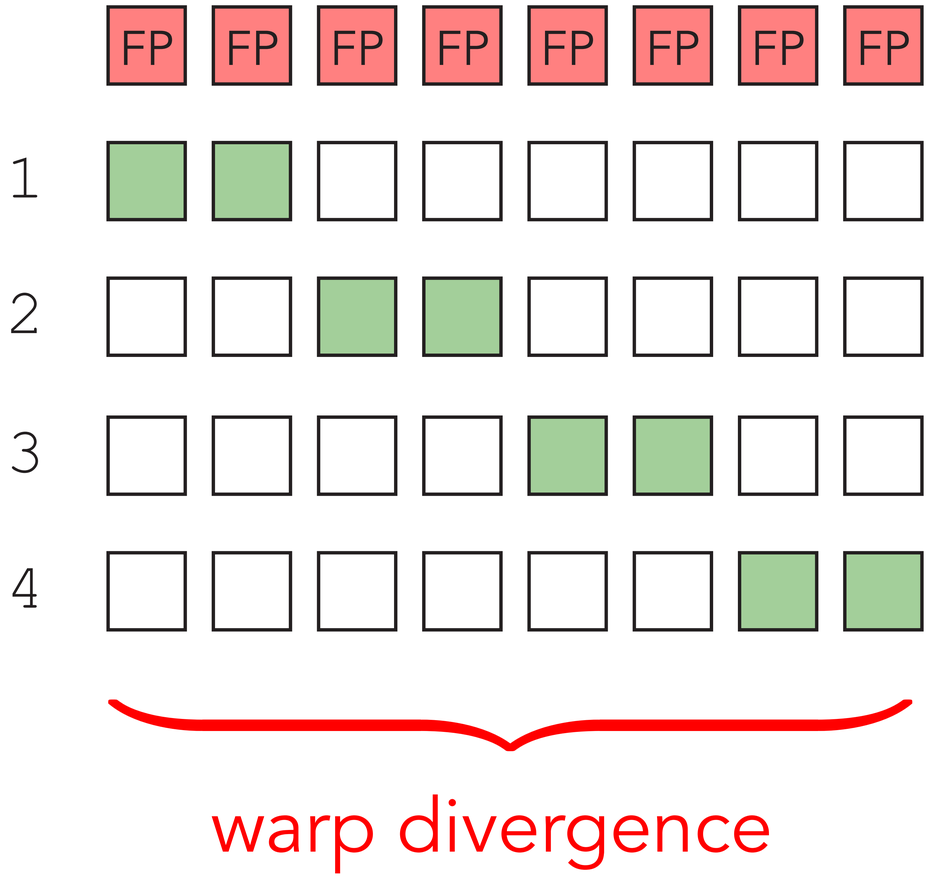

```
if( tid < 2)
  data[tid] += 4;
}
else if( tid < 4) {
  data[tid] -= 4;
}
else if( tid < 6) {
  data[tid] *= 4;
}
else {
  data[tid] /= 4;}
```

Cuda: device abstraction



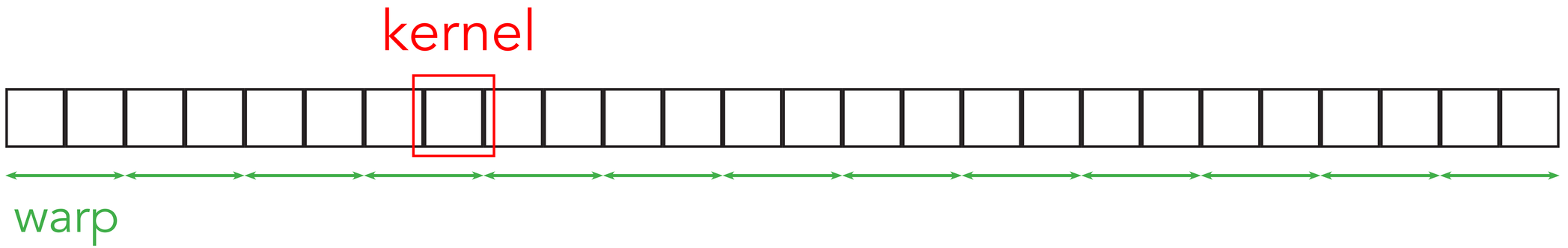

```
if( tid < 2)
  data[tid] += 4;
}
else if( tid < 4) {
  data[tid] -= 4;
}
else if( tid < 6) {
  data[tid] *= 4;
}
else {
  data[tid] /= 4;}
```

Cuda: device abstraction

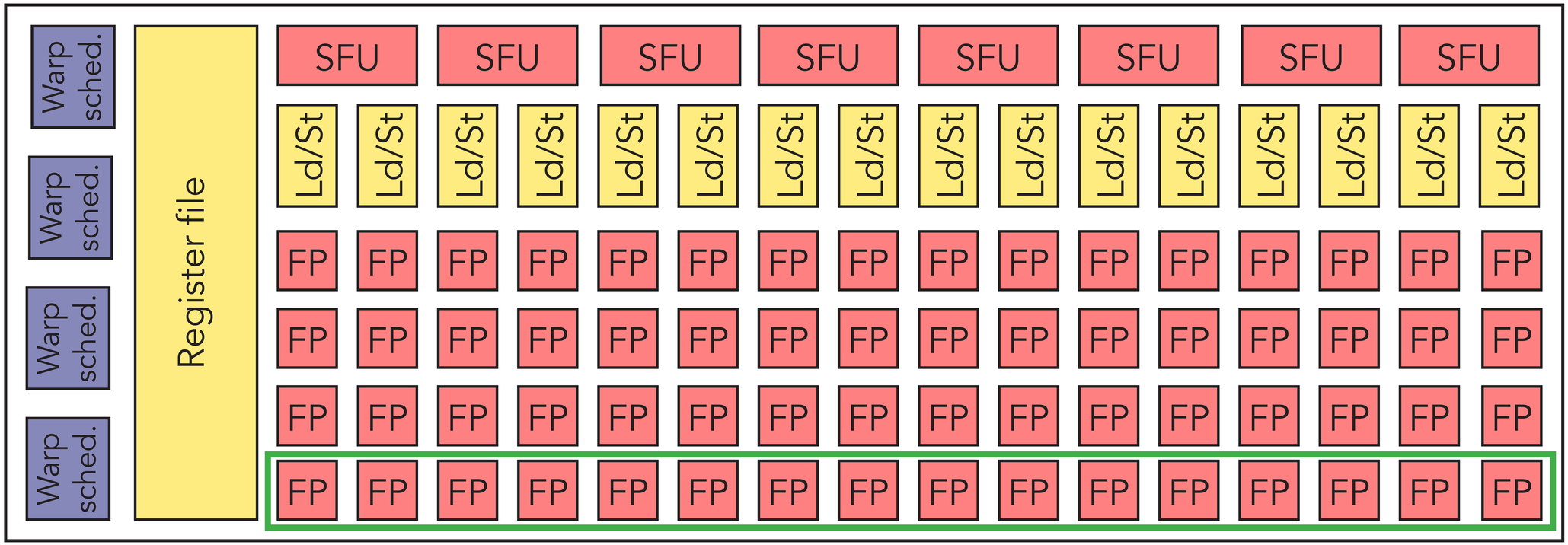



```
if( tid < 2)
  data[tid] += 4;
}
else if( tid < 4) {
  data[tid] -= 4;
}
else if( tid < 6) {
  data[tid] *= 4;
}
else {
  data[tid] /= 4;}
```

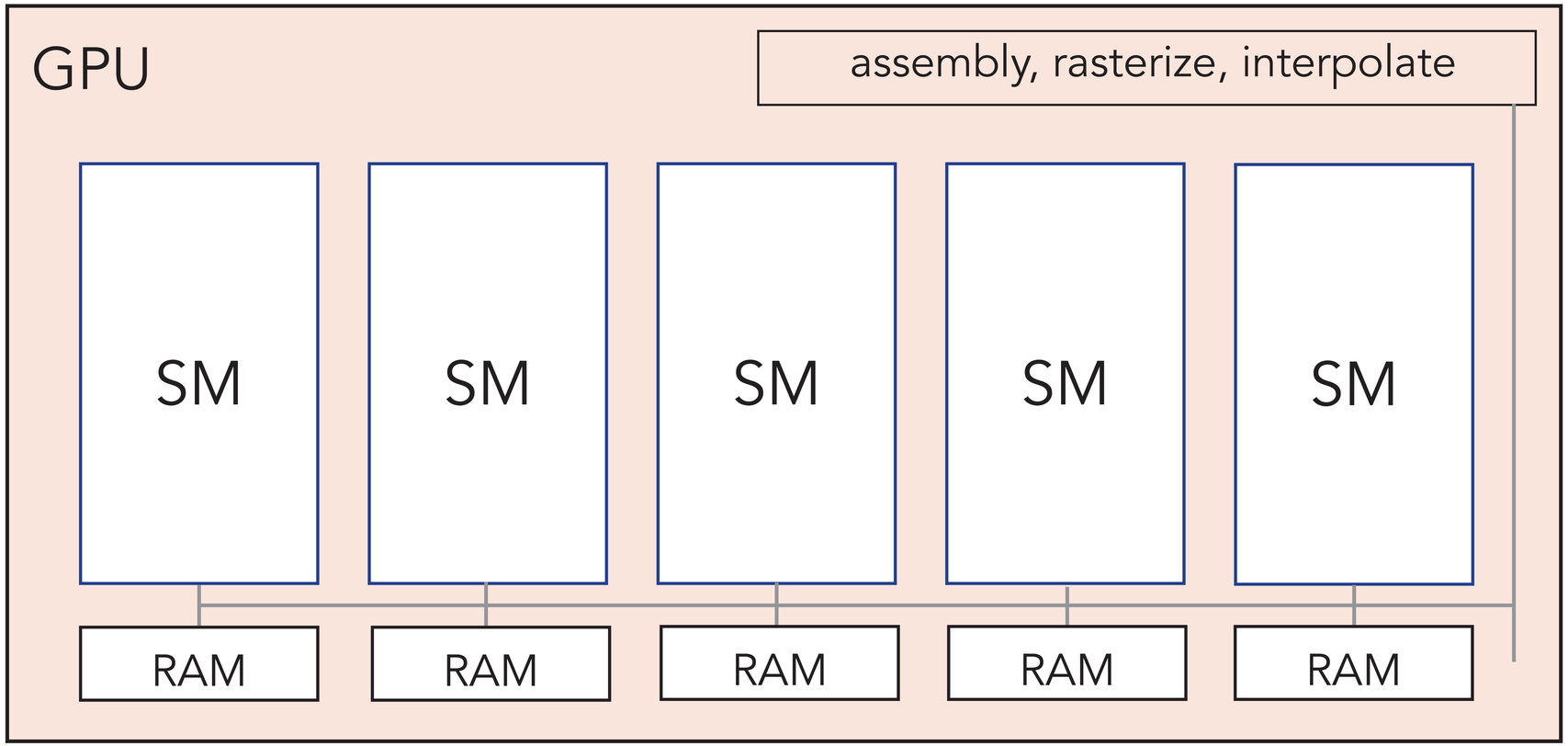
Cuda: device abstraction



Cuda: device abstraction

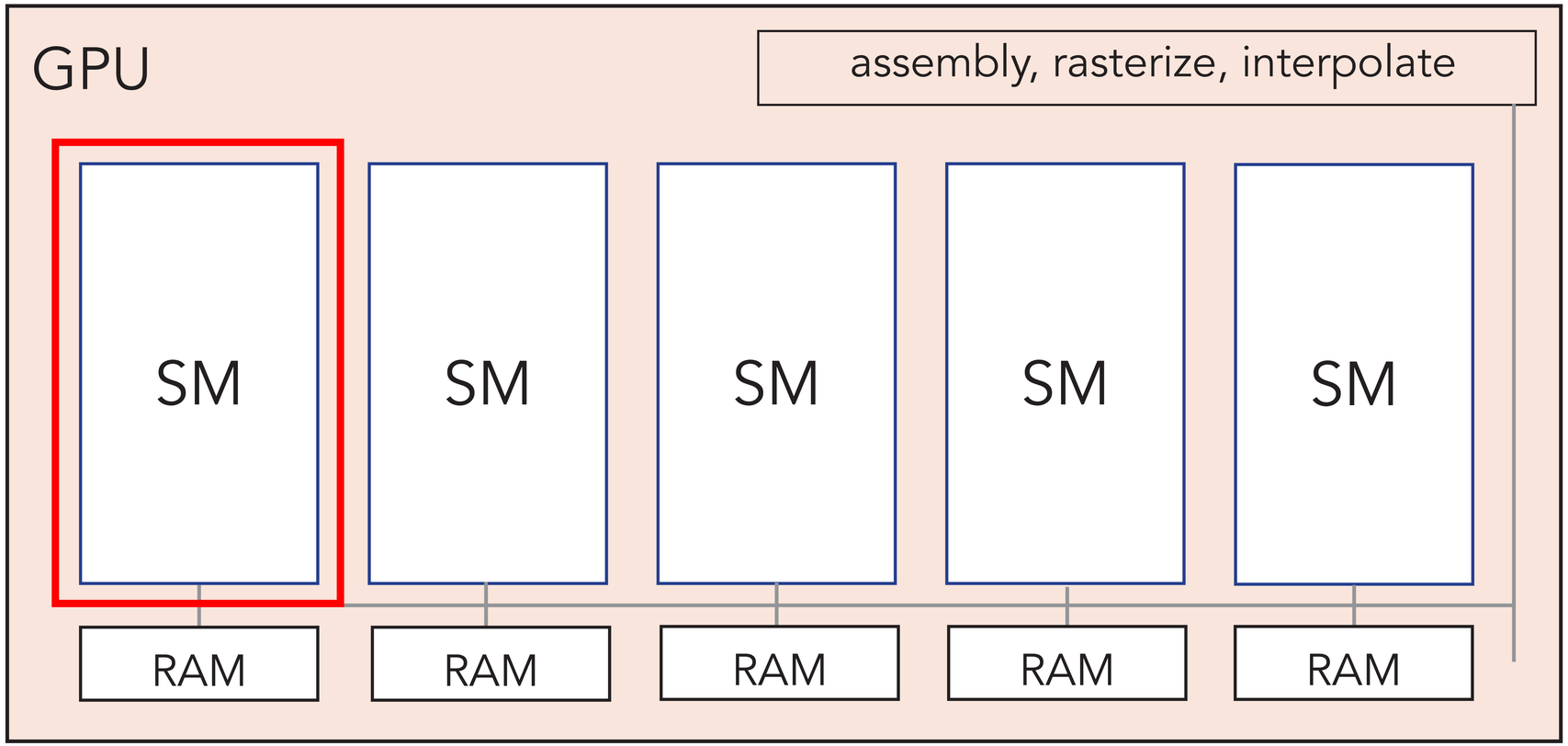


Cuda: device abstraction

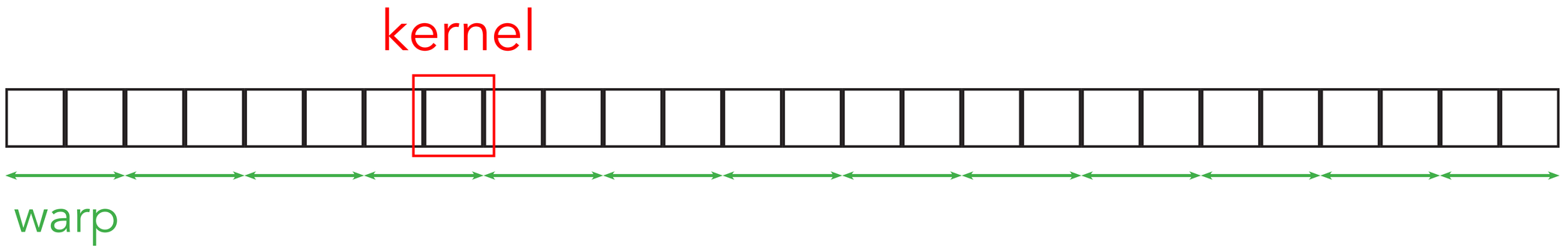


Cuda: device abstraction

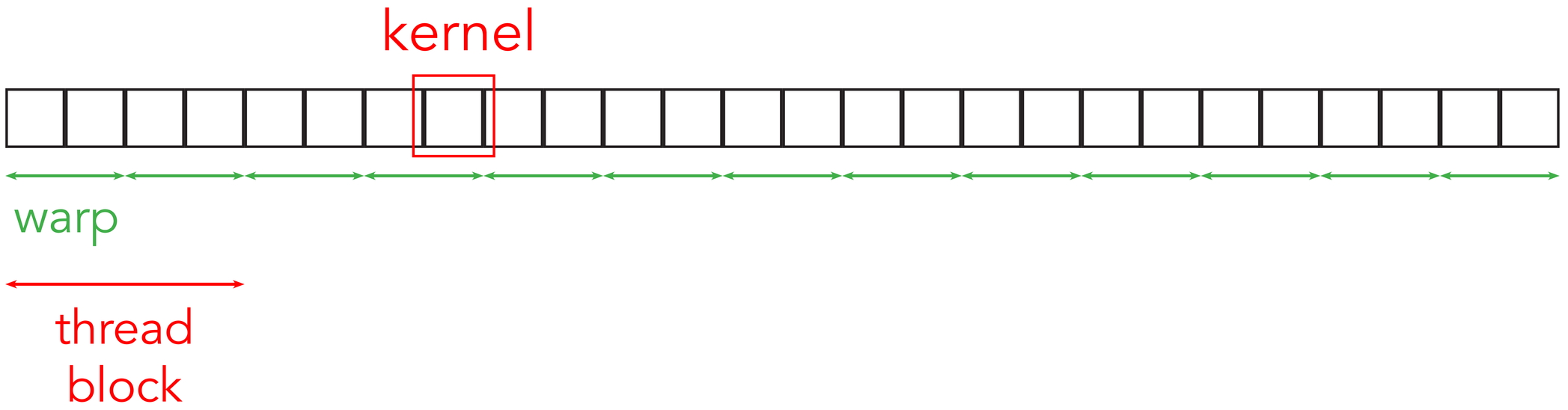
thread block



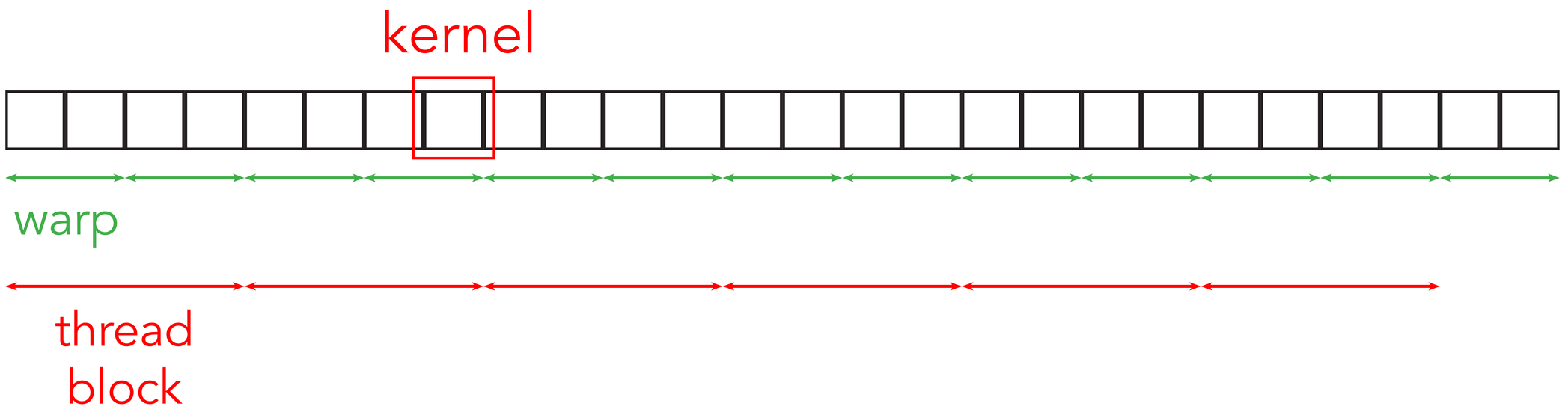
Stream programming model



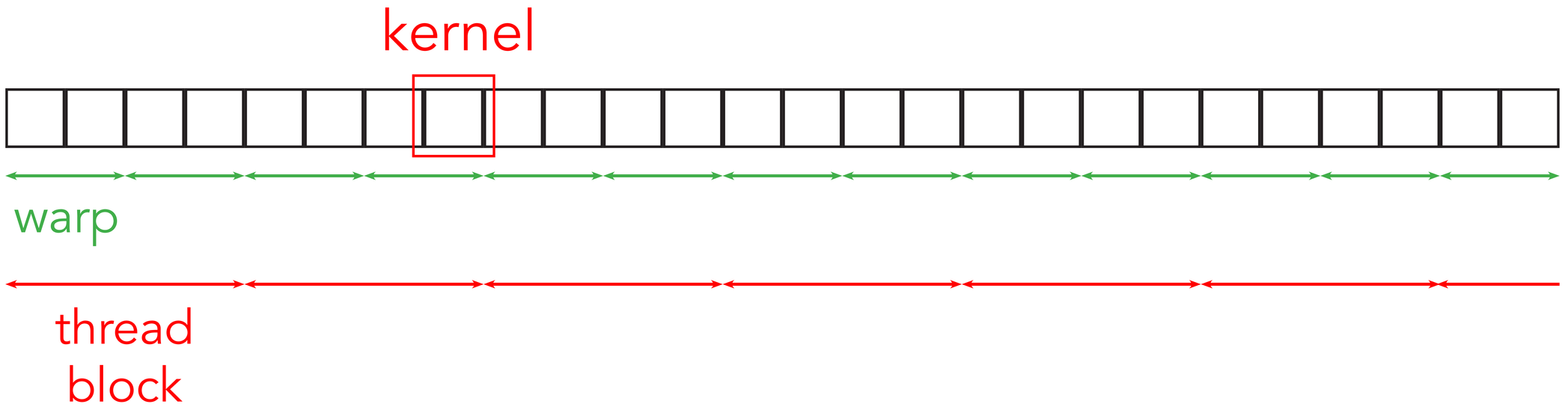
Stream programming model



Stream programming model

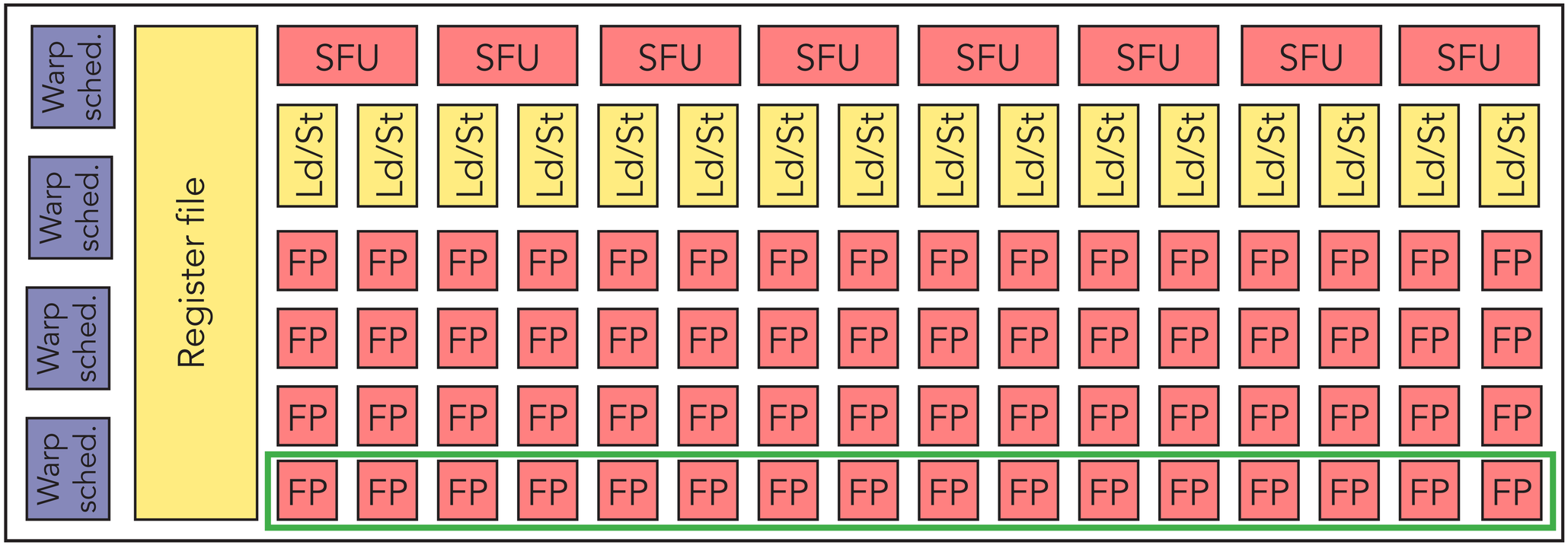


Stream programming model



How many threads / warps per thread block?

Cuda: device abstraction



Cuda: device abstraction

Instruction pipelining:

- ALU latency: 11 cycles
- 1 cycle: 2 instructions for 4 warps

Cuda: device abstraction

Instruction pipelining:

- ALU latency: 11 cycles
- 1 cycle: 2 instructions for 4 warps

=> 44 warps to hide latency

Cuda: device abstraction

Memory pipelining:

- Arithmetic intensity: 60 flops / data
- Memory latency: 300 cycles

Cuda: device abstraction

Memory pipelining:

- Arithmetic intensity: 60 flops / data
- Memory latency: 300 cycles

=> 40 warps to hide latency

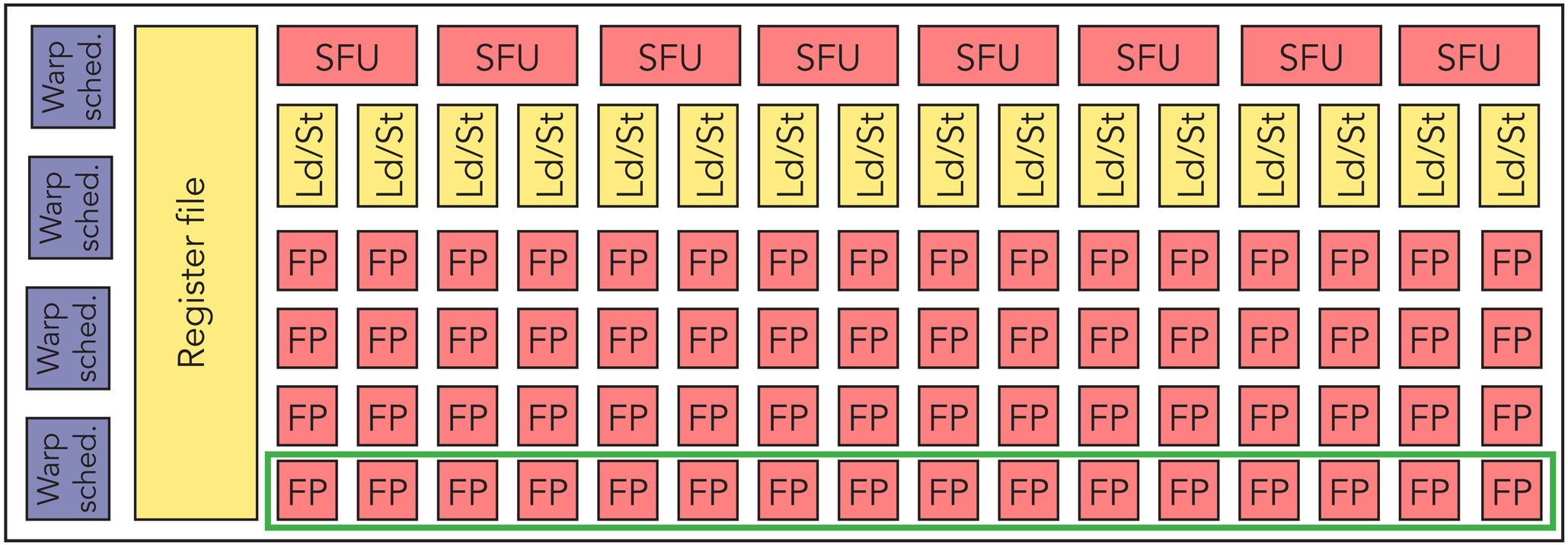
Cuda: device abstraction

Memory pipelining:

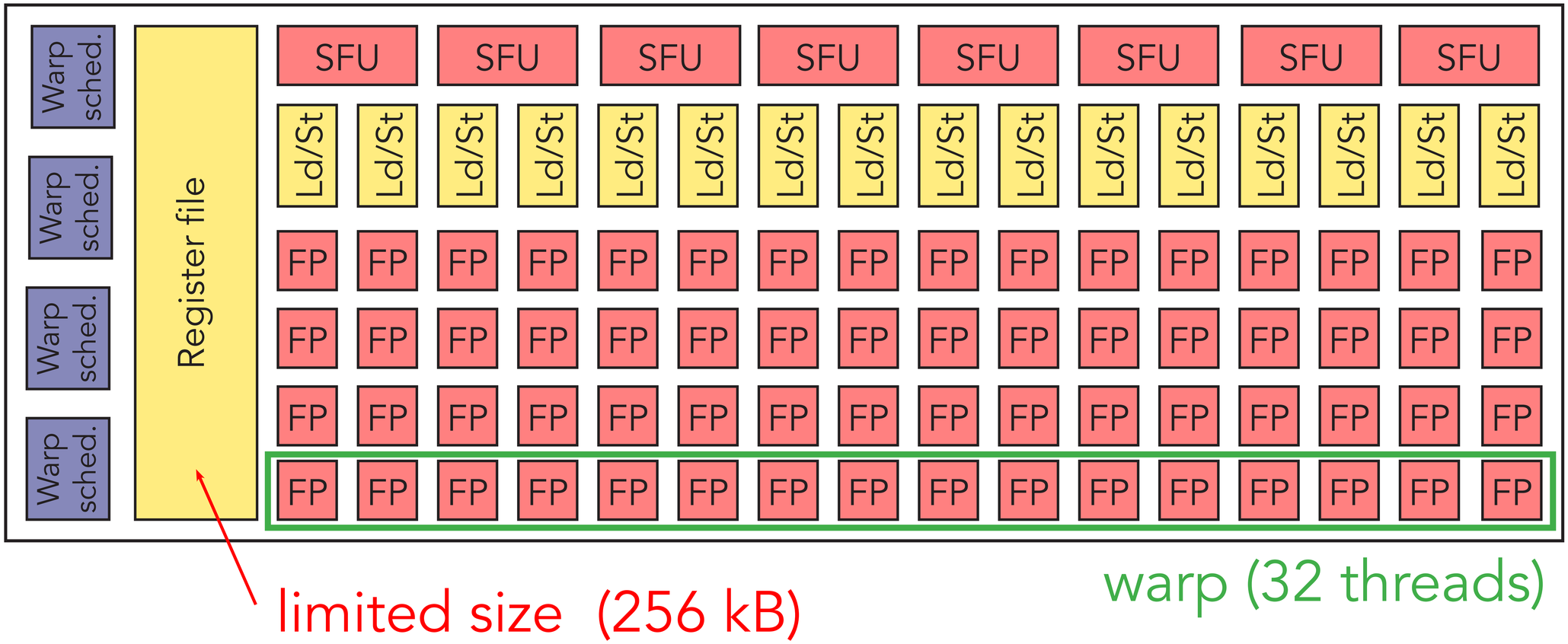
- Arithmetic intensity: 20 flops / data
- Memory latency: 300 cycles

=> 120 warps to hide latency

Cuda: device abstraction



Cuda: device abstraction



Cuda: device abstraction

- Register pressure: limited size of register file
  - More registers required than available leads to spilling of registers to slower memory (RAM, known as local memory)

Cuda: device abstraction

- Register pressure: limited size of register file
  - More registers required than available leads to spilling of registers to slower memory (RAM, then called *local memory*)
  - Example:

$$\frac{256\ \text{kB}}{2^{15}\ \text{threads}} = 2\ \text{registers}$$

Cuda: device abstraction

- Register pressure: limited size of register file
  - More registers required than available leads to spilling of registers to slower memory (RAM, then called *local memory*)
  - Example:

$$\frac{256\ \text{kB}}{2^{15}\ \text{threads}} = 2\ \text{registers}$$

number of threads per SM is limited (2048)

Cuda: device abstraction

- Occupancy:

$$\mathcal{O} = \frac{\#\ \text{threads / SM}}{\text{max.}\ \#\ \text{threads / SM}}$$

Cuda: device abstraction

- Occupancy:

$$\mathcal{O} = \frac{\#\ \text{threads}\ /\ \text{SM}}{\text{max.}\ \#\ \text{threads}\ /\ \text{SM}}$$

  - Maximize subject to available resources

Cuda: device abstraction

- Occupancy examples: 32 registers / thread

Cuda: device abstraction

- Occupancy examples: 32 registers / thread
  - 256 kbyte / (32 * 4 byte) = 2048
  - $\mathcal{O}$ = 1.0

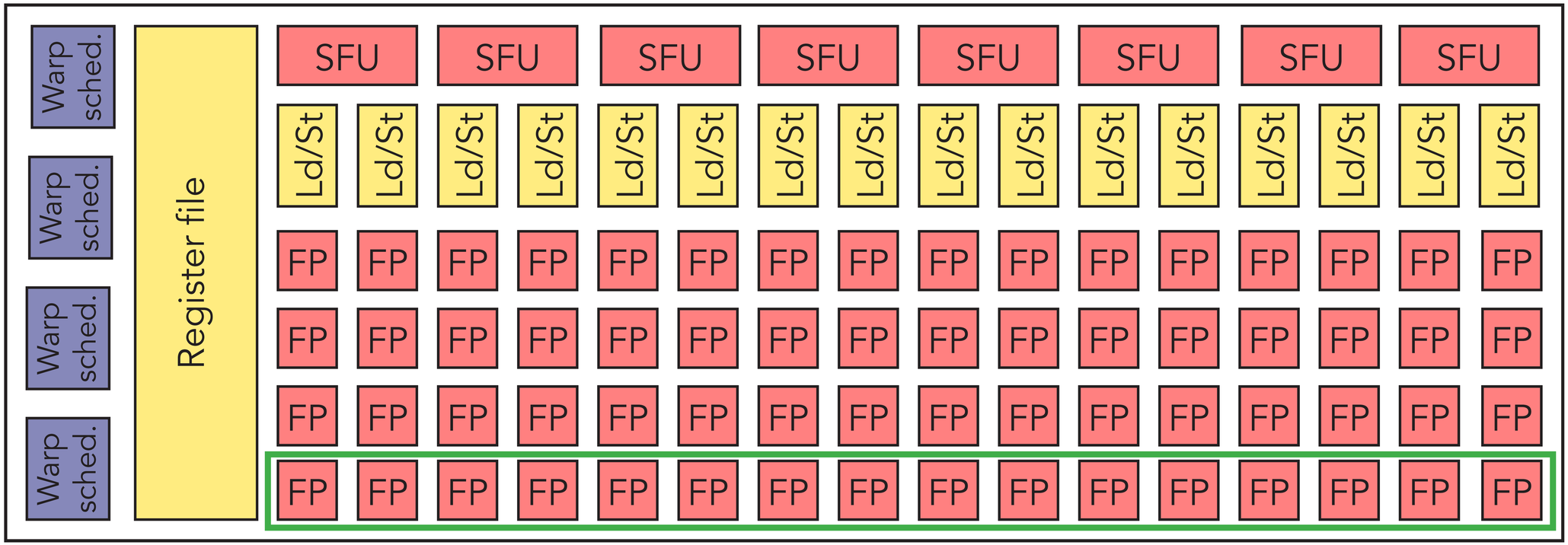
Cuda: device abstraction

- Occupancy examples: 32 registers / thread
  - 256 kbyte / (32 * 4 byte) = 2048
  - $\mathcal{O}$ = 1.0
- Occupancy examples: 64 registers / thread

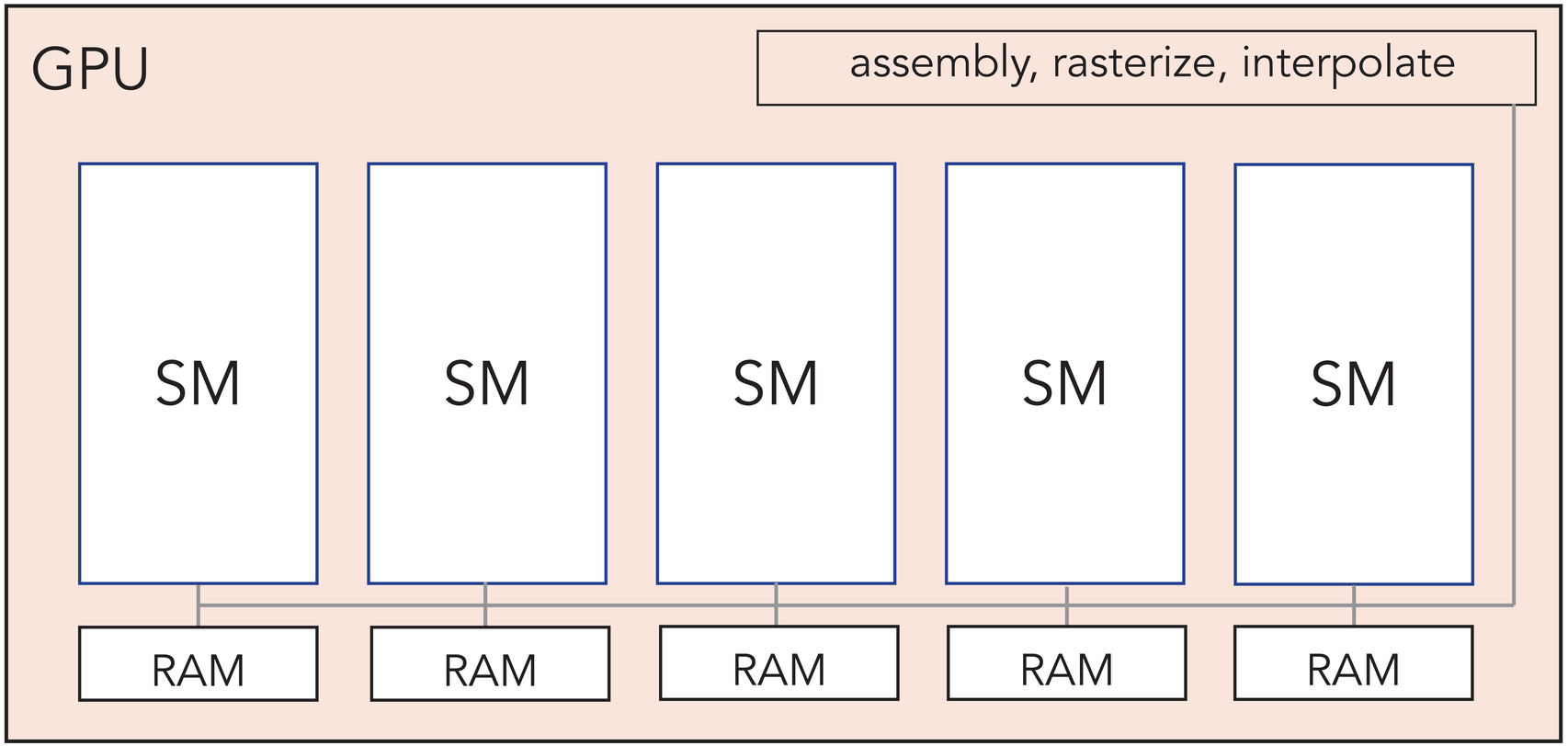
Cuda: device abstraction

- Occupancy examples: 32 registers / thread
  - 256 kbyte / (32 * 4 byte) = 2048
  - $\mathcal{O}$ = 1.0
- Occupancy examples: 64 registers / thread
  - 256 kbyte / (64 * 4 byte) = 1024
  - $\mathcal{O}$ = 0.5

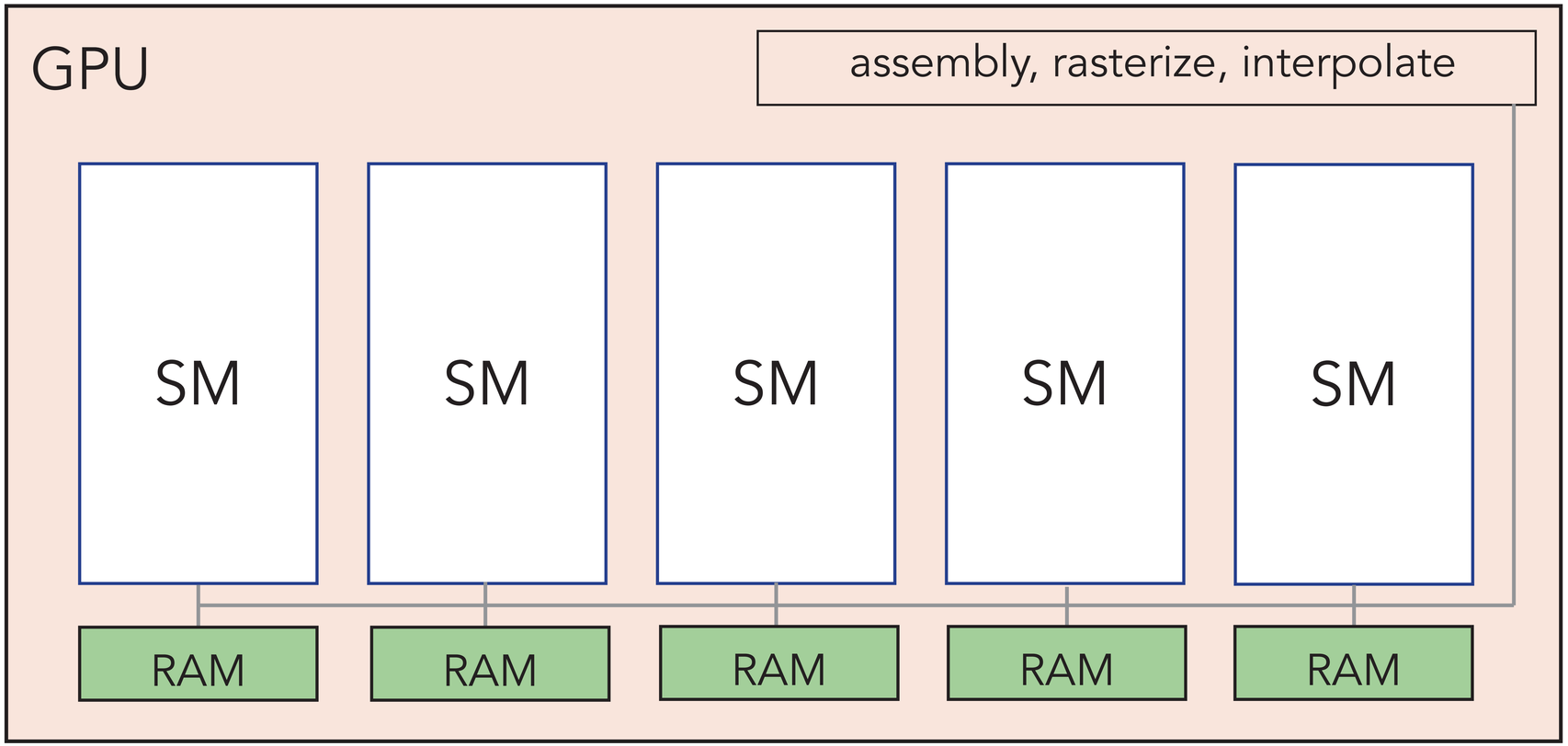
Cuda: device abstraction



Cuda memory

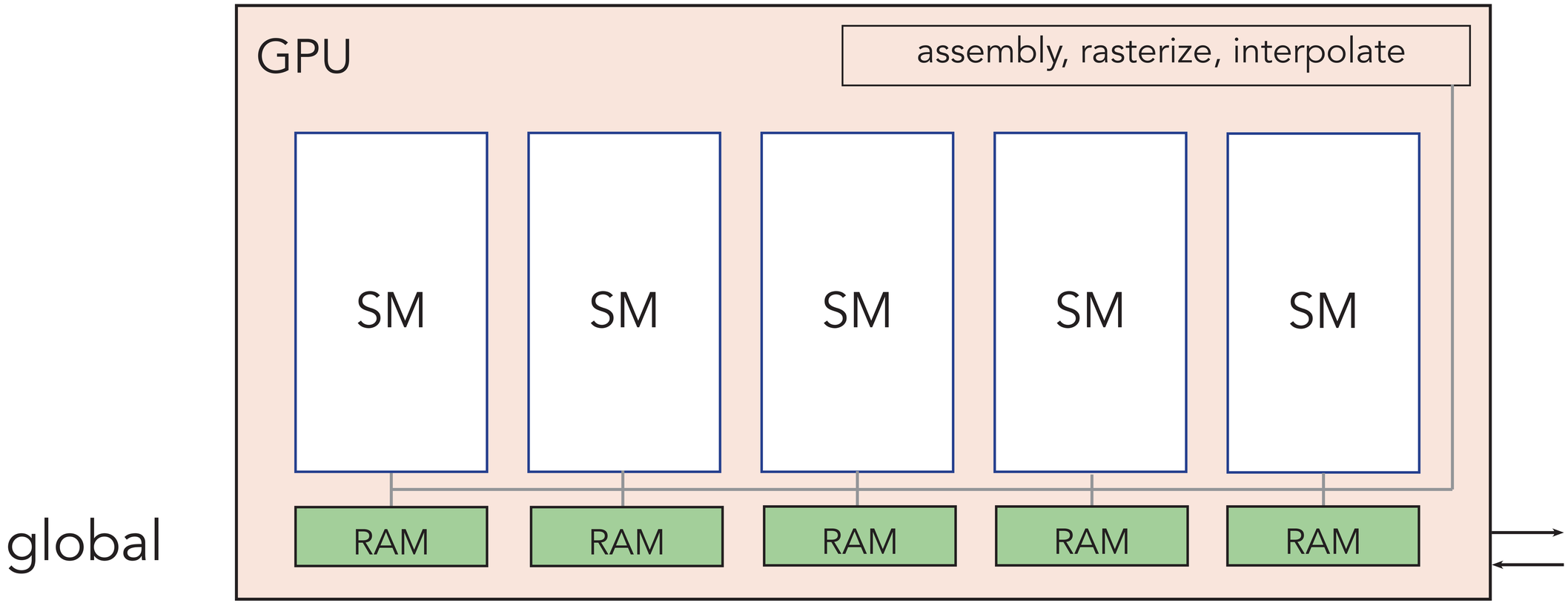


Cuda memory

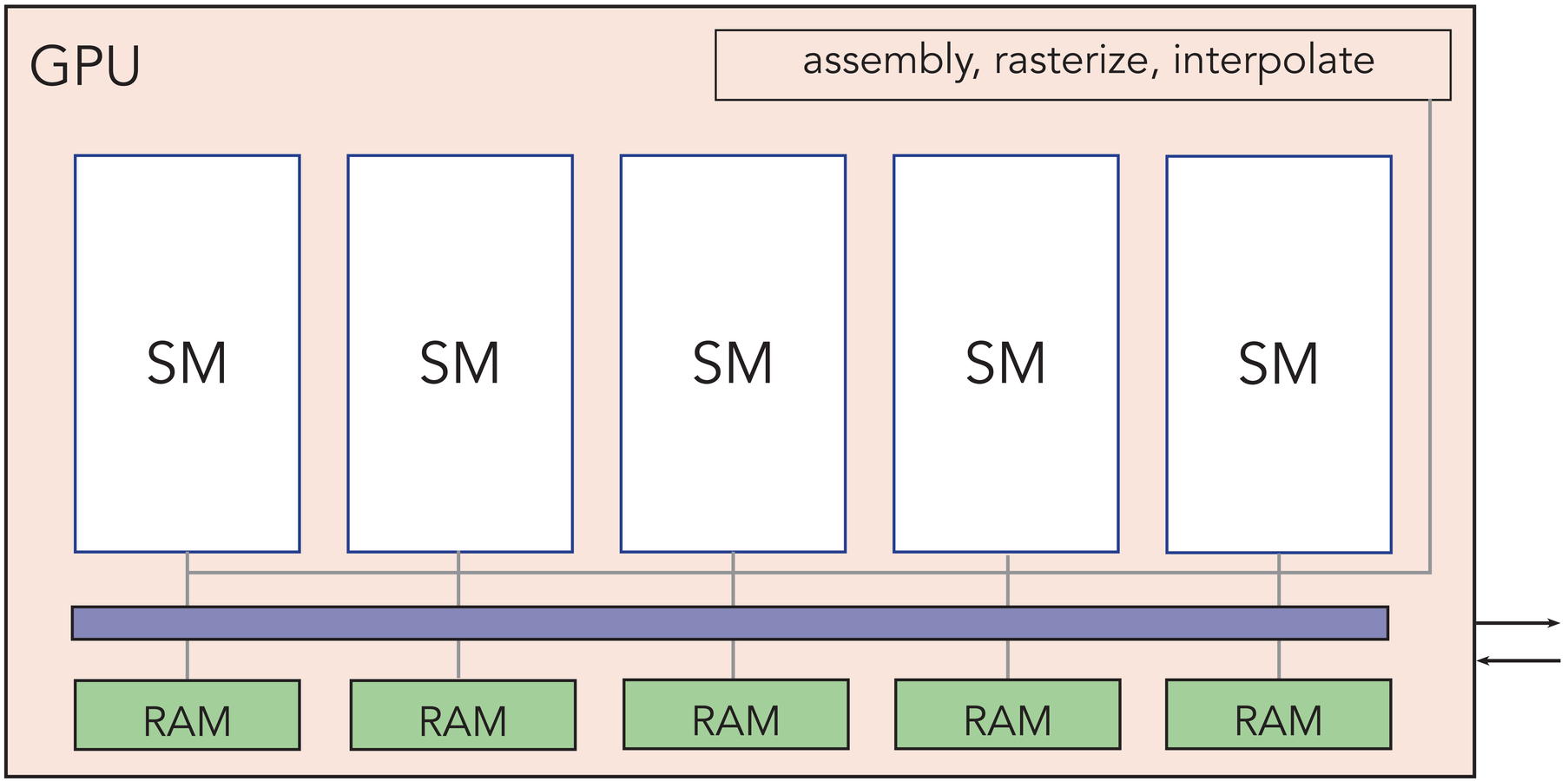


global

Cuda memory



Cuda memory

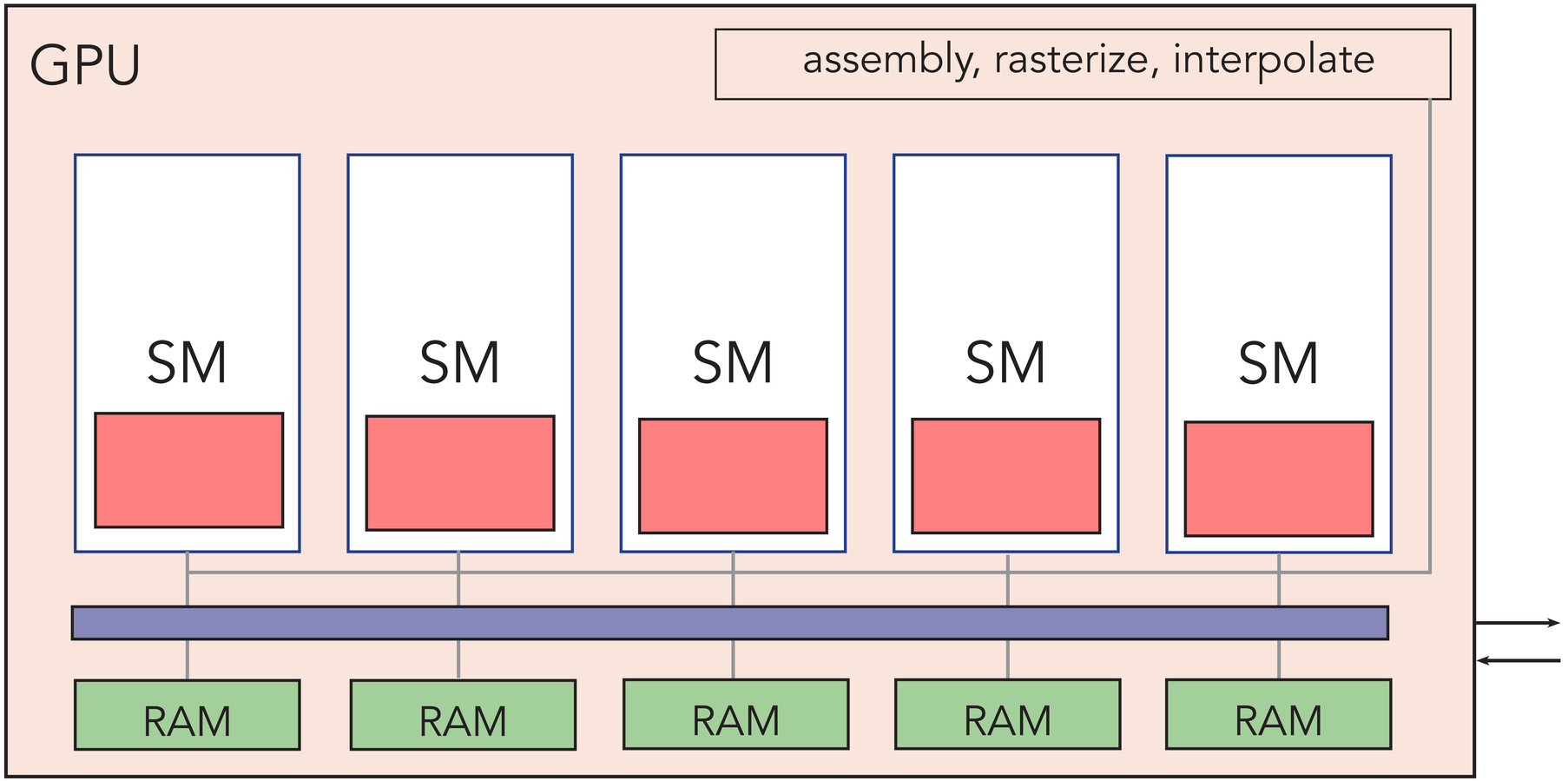


Cuda memory

shared mem/
L1 cache

L2 cache

global



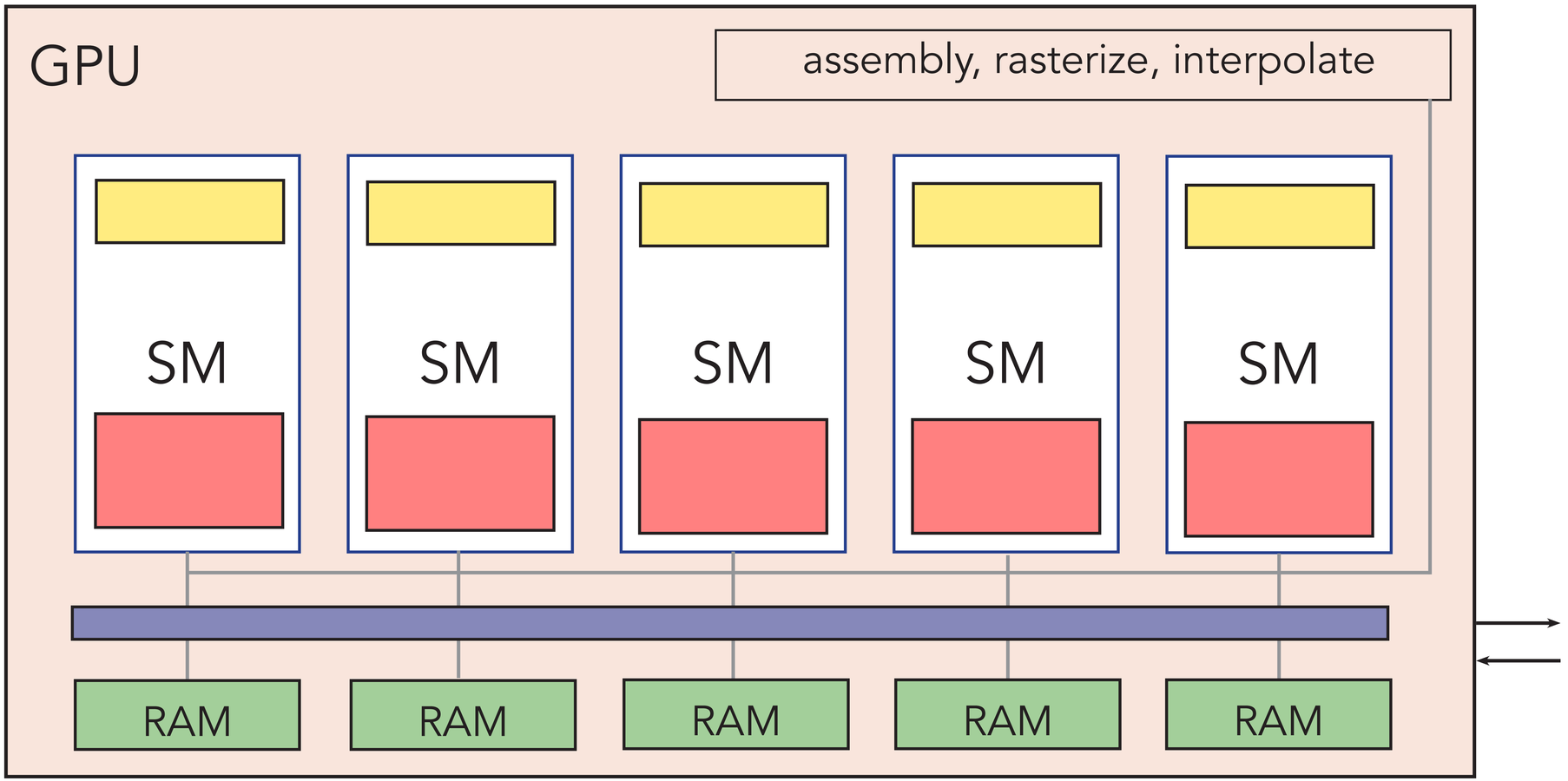
Cuda memory

register

shared mem/
L1 cache

L2 cache

global



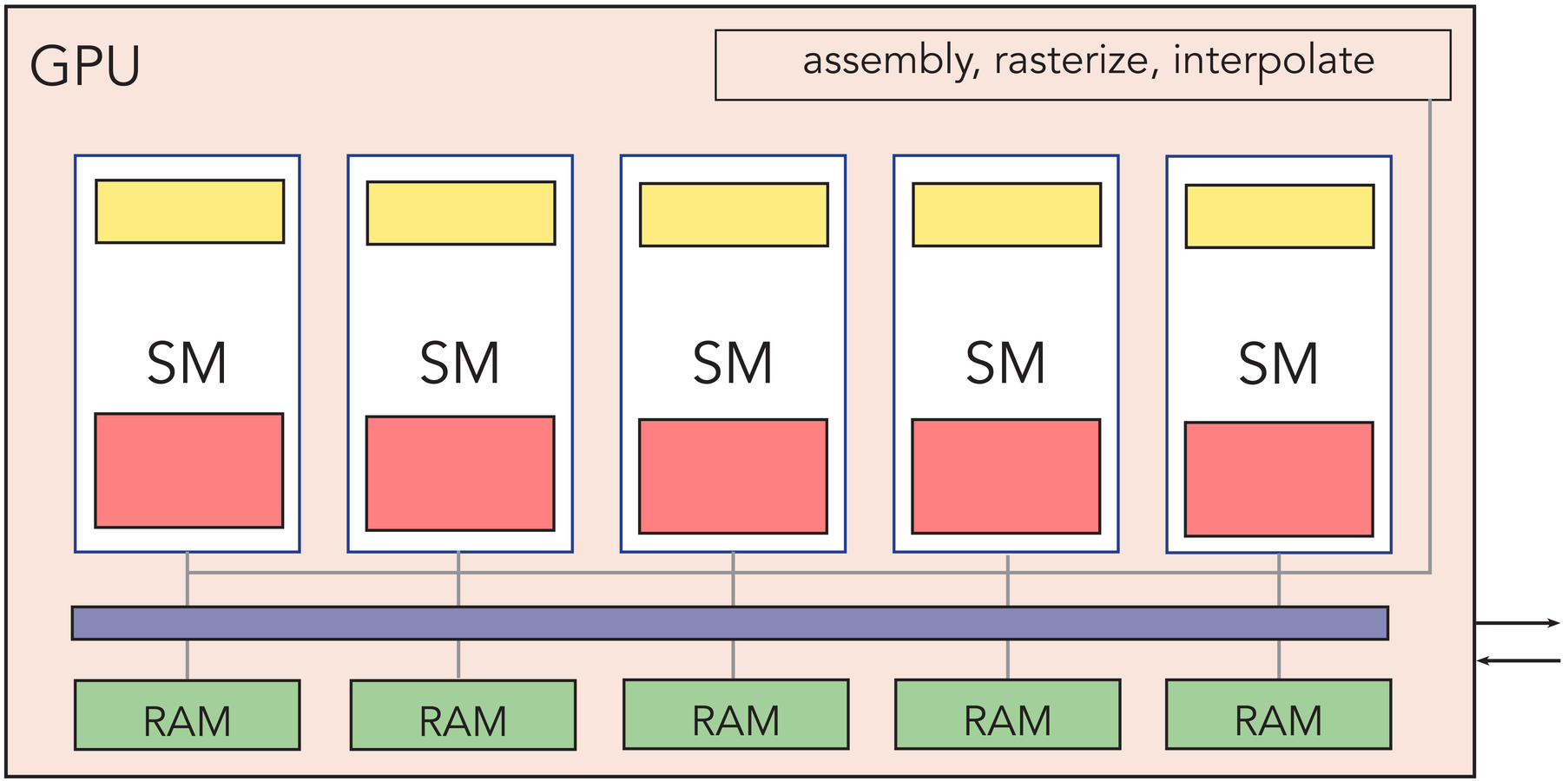
Cuda memory

register

shared mem/
L1 cache

L2 cache

global / local

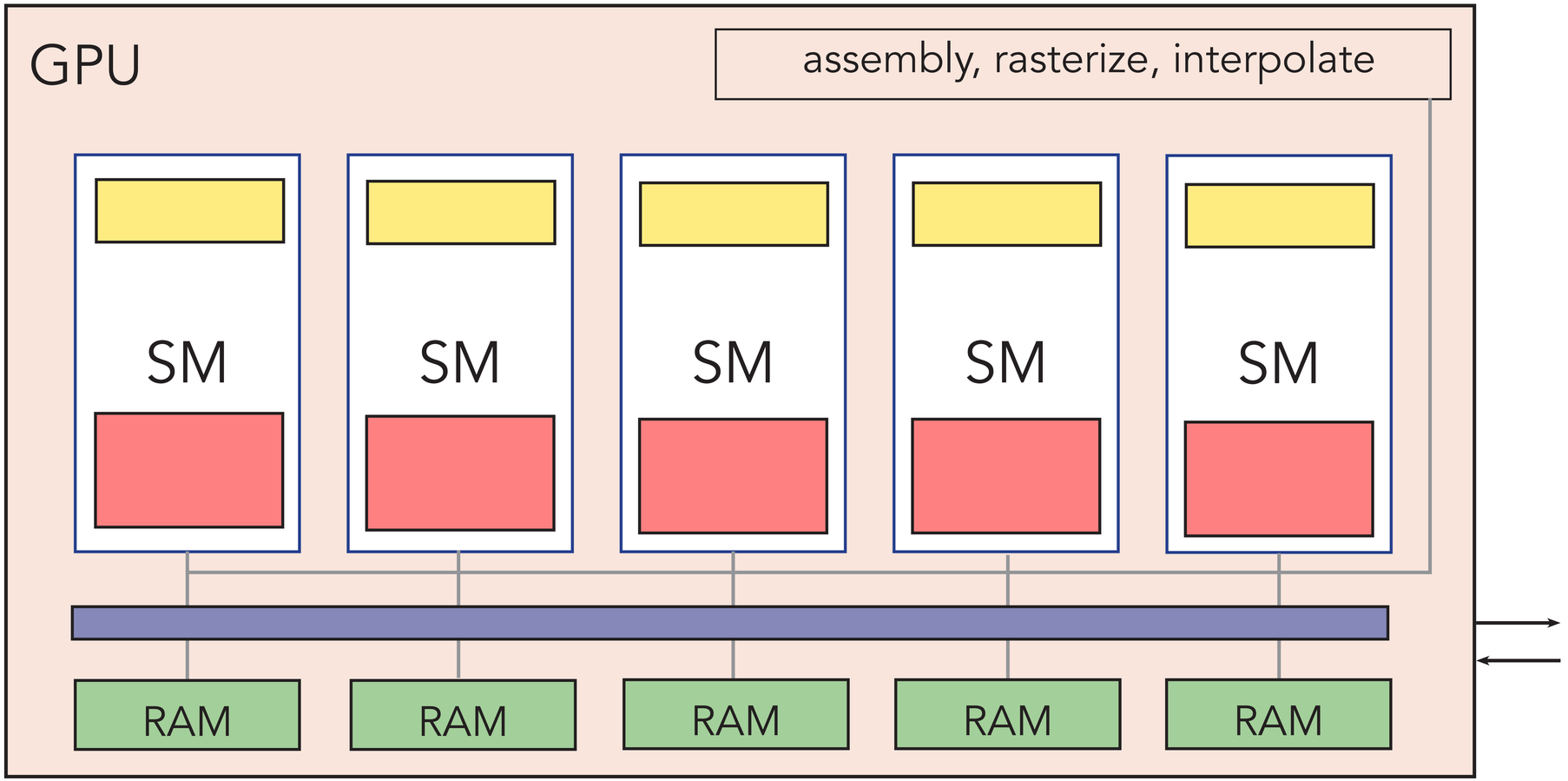


Cuda memory

1 cycle

10 cycles

300 cycles

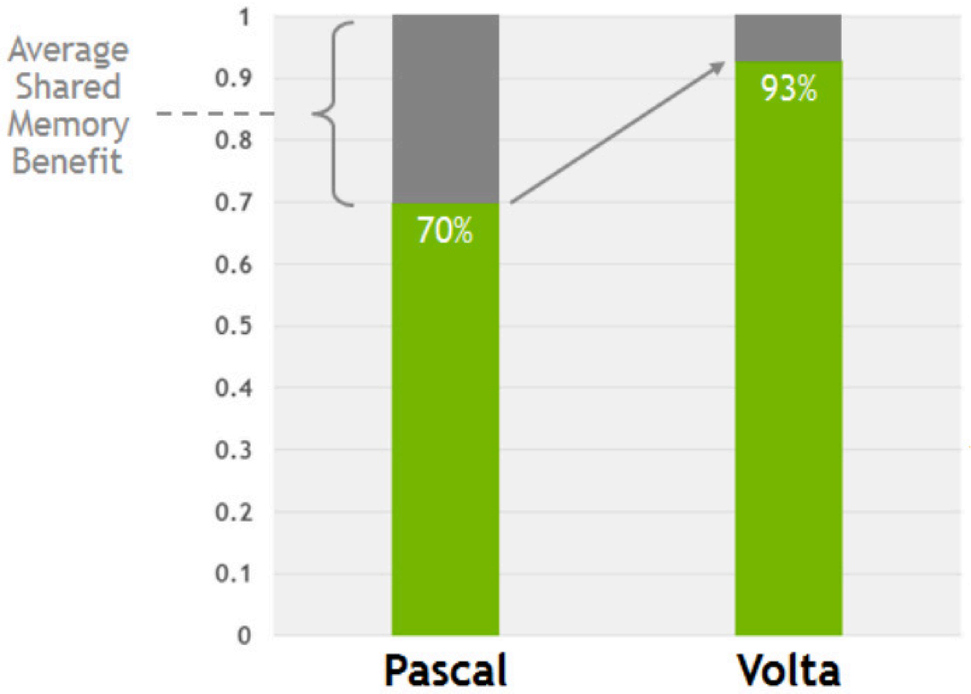


Cuda memory

- Shared memory
  - Configurable: L1 cache, user managed cache, both

Cuda memory

- Shared memory
  - Configurable: L1 cache, user managed cache, both



https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

Cuda memory

- Shared memory
  - Configurable: L1 cache, user managed cache, both
  - Shared by all threads in a thread block
  - 16 kB - 48 kB

Cuda memory

- Shared memory
  - Configurable: L1 cache, user managed cache, both
  - Shared by all threads in a thread block
  - 16 kB - 48 kB

can be used for
communication / synchronization

Cuda memory

```
__global__ void
myKernel( float* gdata) {

  __shared__ float sdata[1024];
  sdata[threadIdx.x] = gdata[threadIdx.x];
  __syncthreads();

  // process data in shared memory
  ...

  // write data back from shared to global mem
}
```

Cuda memory

```
__global__ void
myKernel( float* gdata) {

  __shared__ float sdata[1024];
  sdata[threadIdx.x] = gdata[threadIdx.x];
  __syncthreads();

  // process data in shared memory
  ...

  // write data back from shared to global mem
}
```

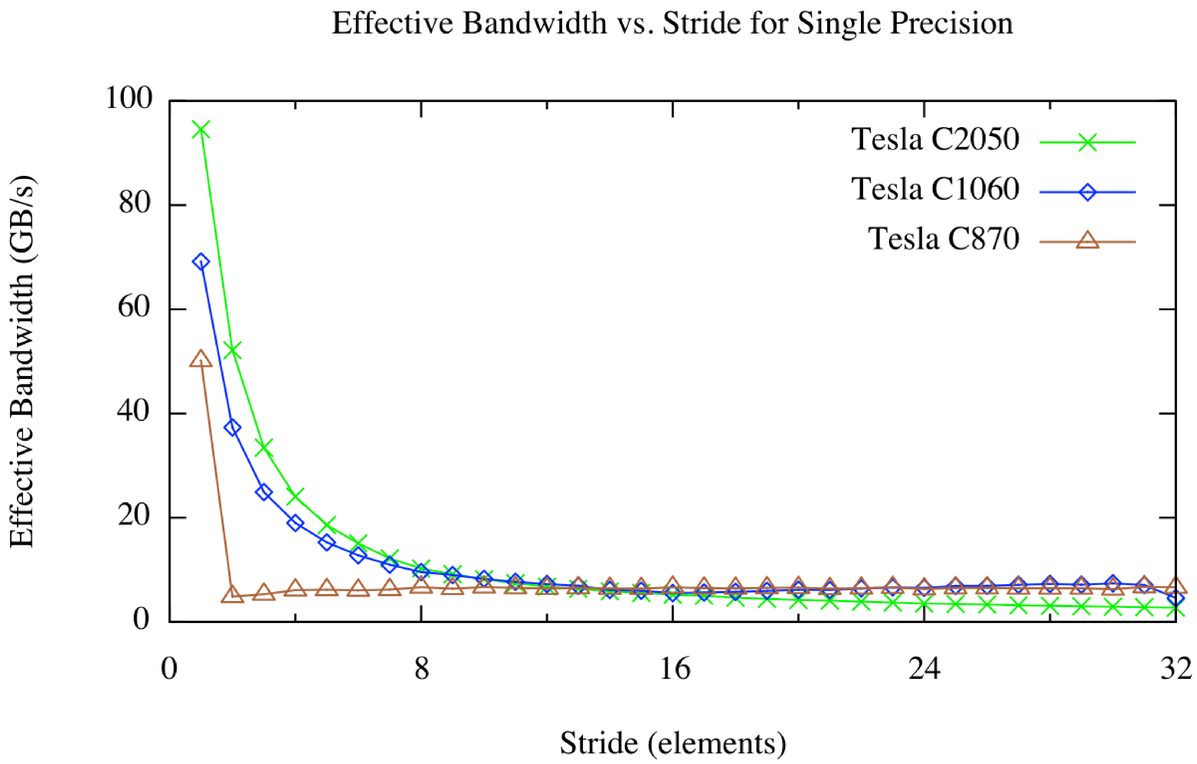
affects occupancy

Cuda memory

- Shared memory: latency affected by bank conflicts
  - Memory module consists of 32 banks
  - 1 LD/ST per cycle per bank
  - Multiple accesses to the same bank from multiple threads in a warp result in serialization

Cuda memory

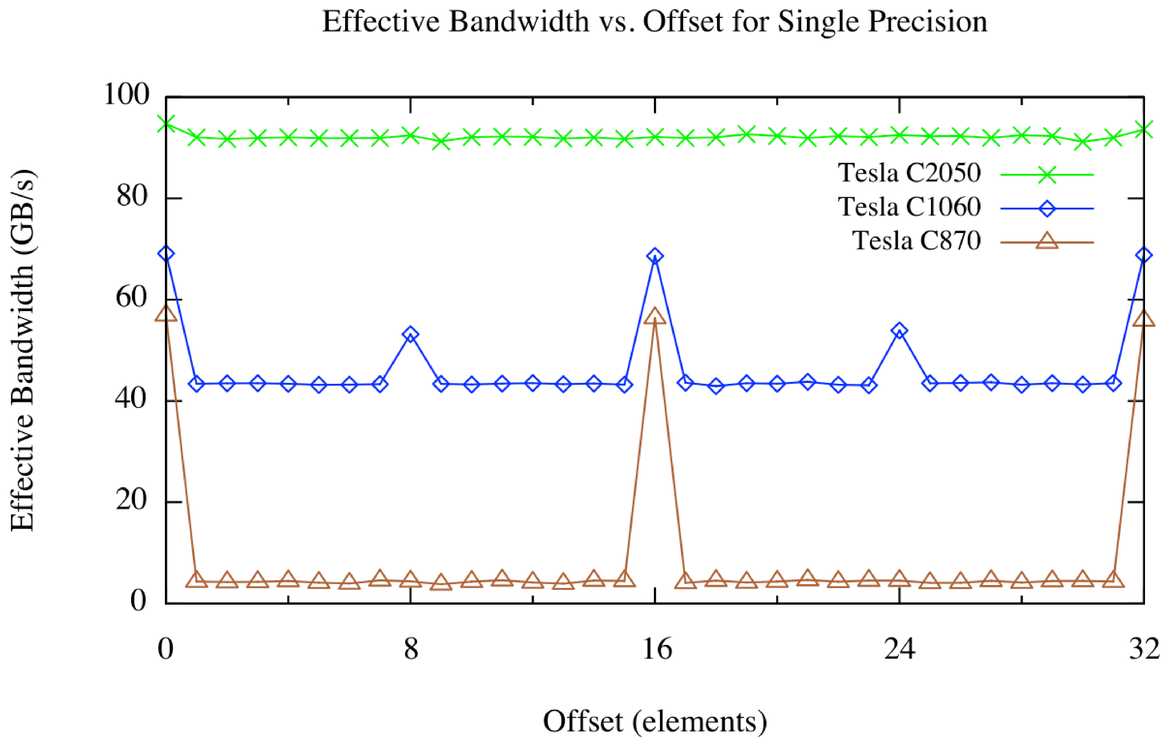
- Global memory: latency affected by coalescing



https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

Cuda memory

- Global memory: latency affected by coalescing



https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

Cuda memory

- Constant memory
  - Program parameters that can not be compiled into instruction stream
  - In RAM but separate cache
  - 64 kB

Cuda memory

- Texture memory
  - Useful for 2D data, e.g. images
  - Better caching behaviour for 2D data
  - Free linear / bilinear interpolation of data
  - Stored in global memory but in swizzled format (maximizes 2D caching performance)

Cuda device capabilities

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	255[1]
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/ 48 KB	96 KB	64 KB	Configurable up to 96 KB

[1] The per-thread program counter (PC) that forms part of the improved SIMT model typically requires two of the register slots per thread.

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

Further reading

- D. Kirk and W. -m. Hwu, Programming massively parallel processors. Elsevier/Morgan Kaufmann, 2013, Ch. 1.
- http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
- http://www.nvidia.de/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf
- https://devblogs.nvidia.com/parallelforall/
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/
- http://docs.nvidia.com/cuda/cuda-runtime-api/