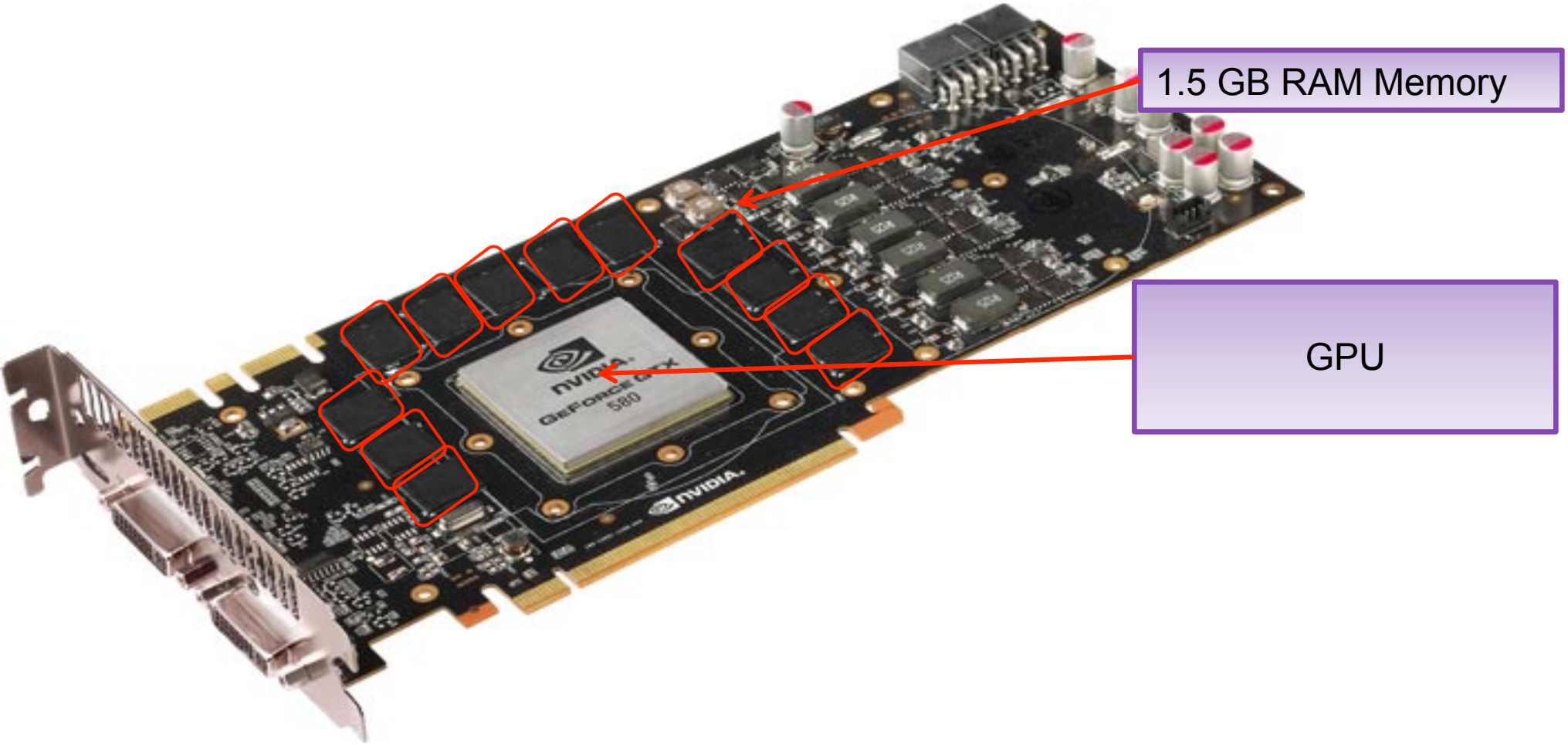
Seminar 4

CUDA

and

Efficient GPU Programming

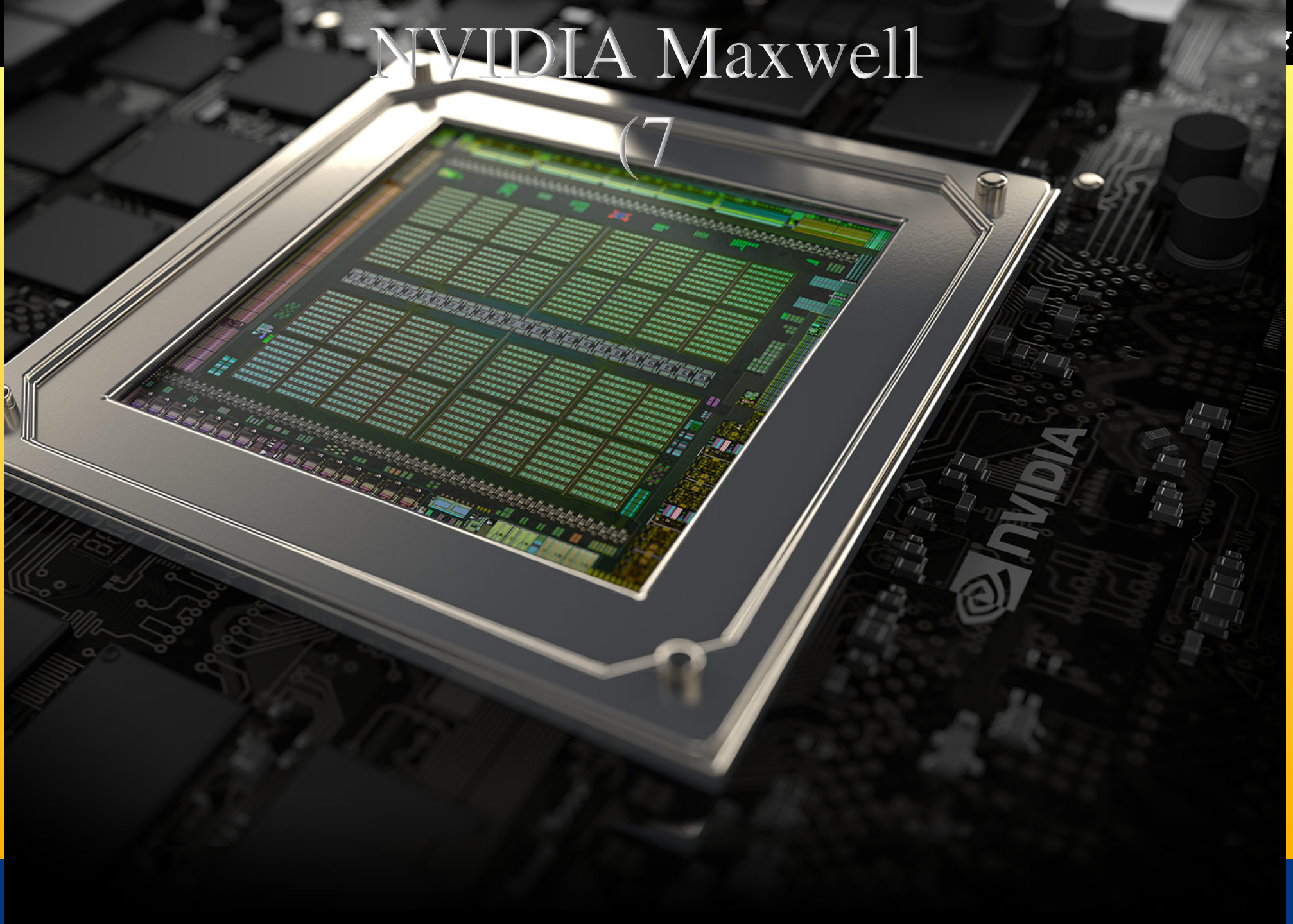
Graphics Processing Unit - GPU

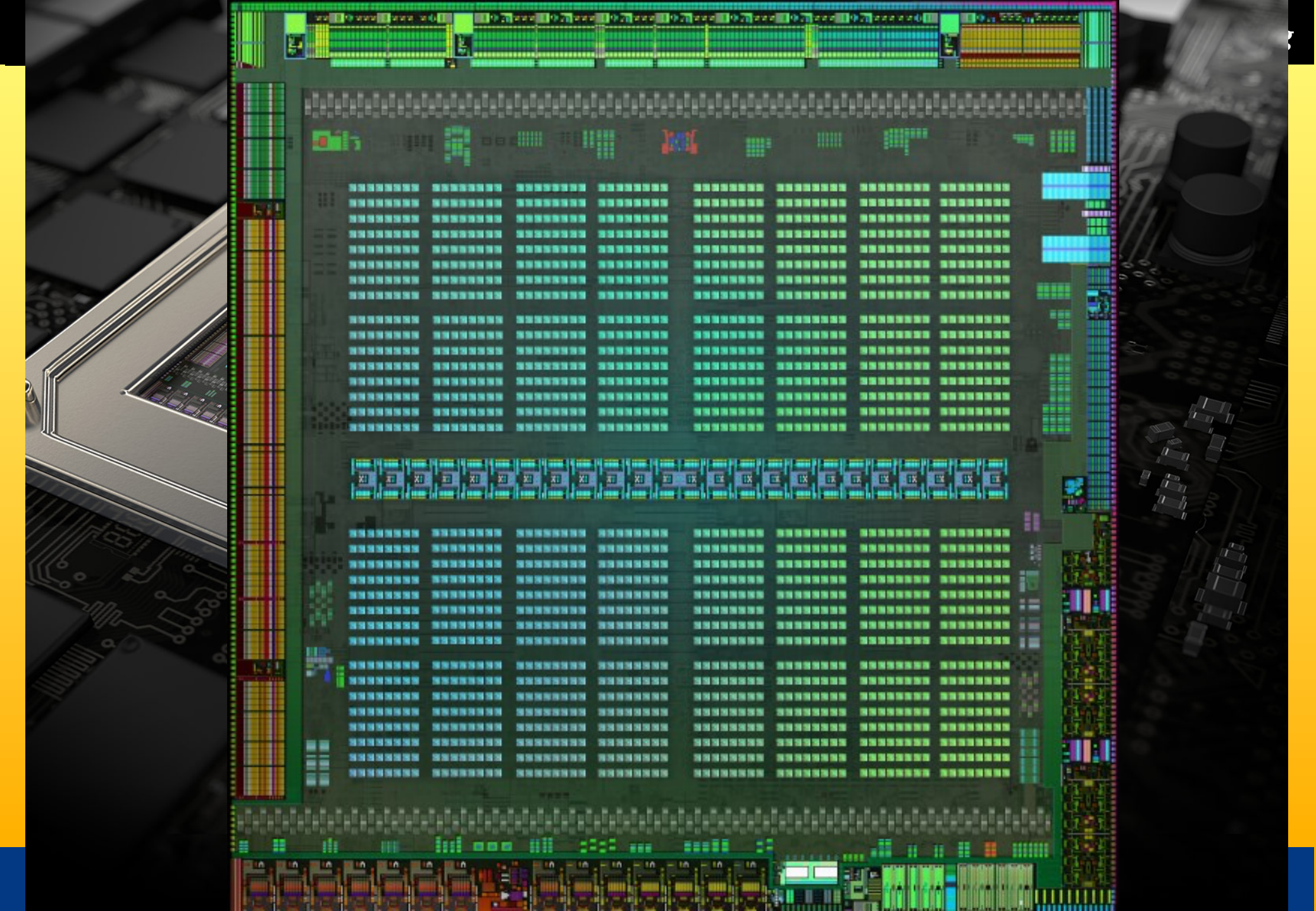


- NVIDIA Geforce GTX 580

NVIDIA Maxwell

(7

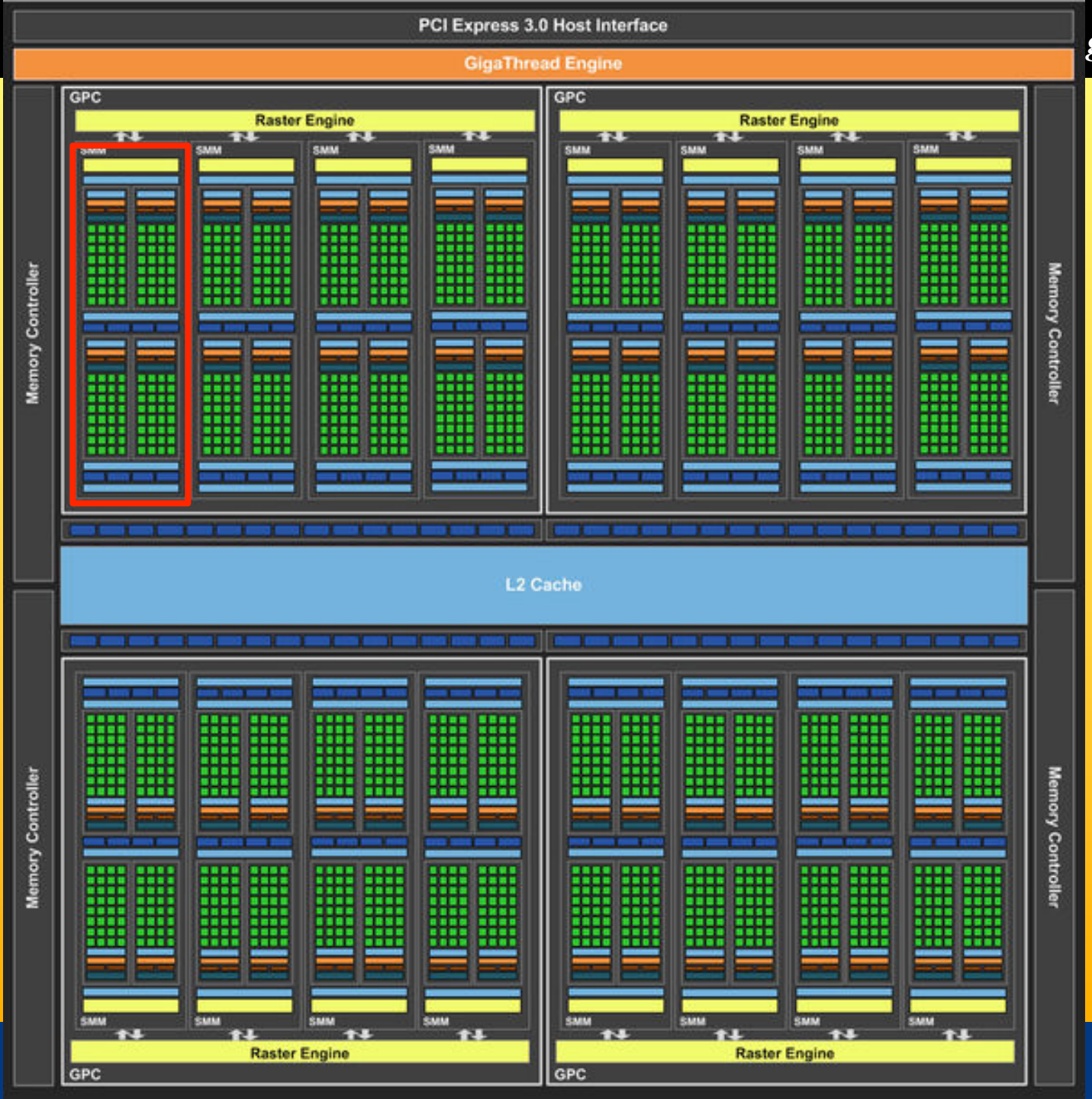




16 SMMs (“Cores”)
2MB L2 cache
64 color outputs/ clock (i.e., 64 ROPs)

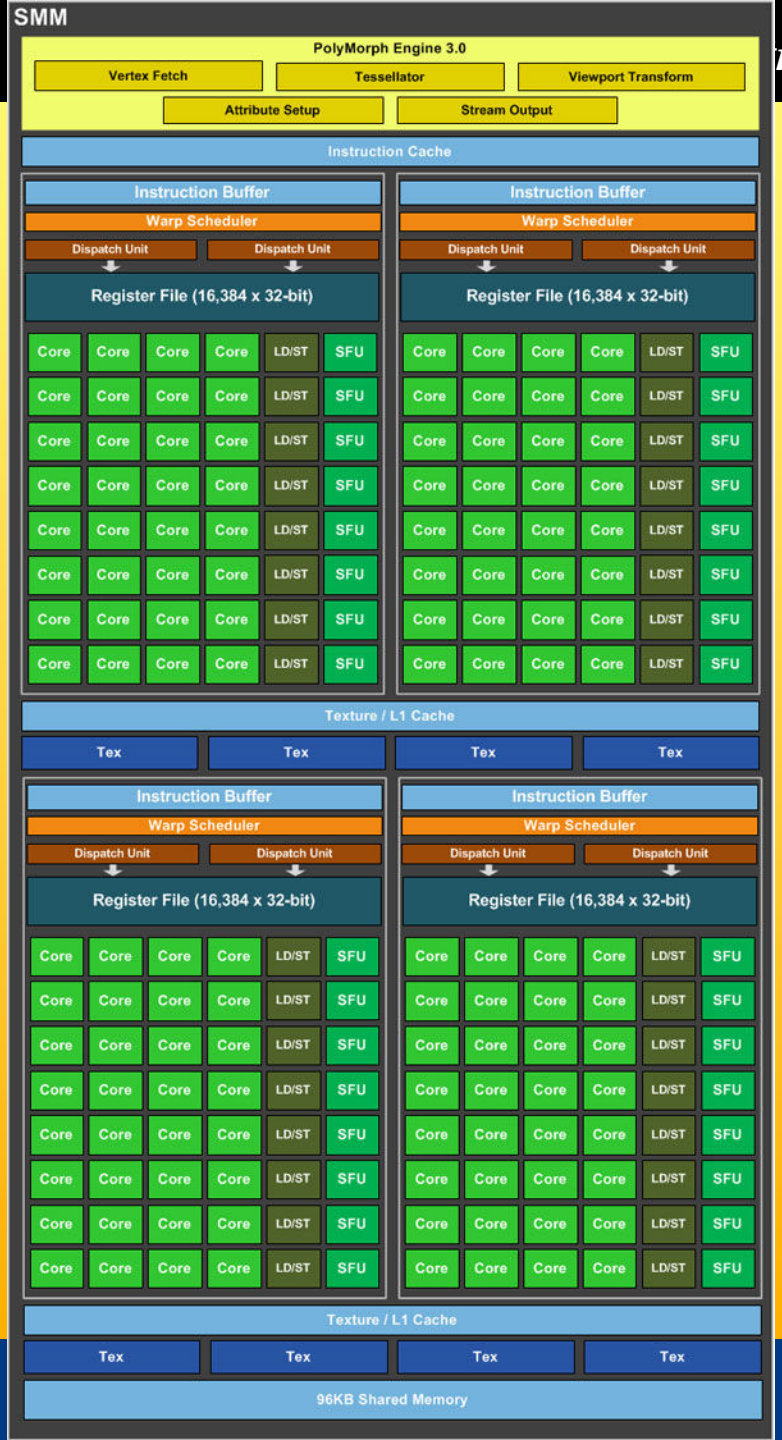
Each SMM:

- 128 ALUs
- 96KB L1 cache
- 8 TexUnits
- 32 Load/Store units for access to global memory



Each SMM:

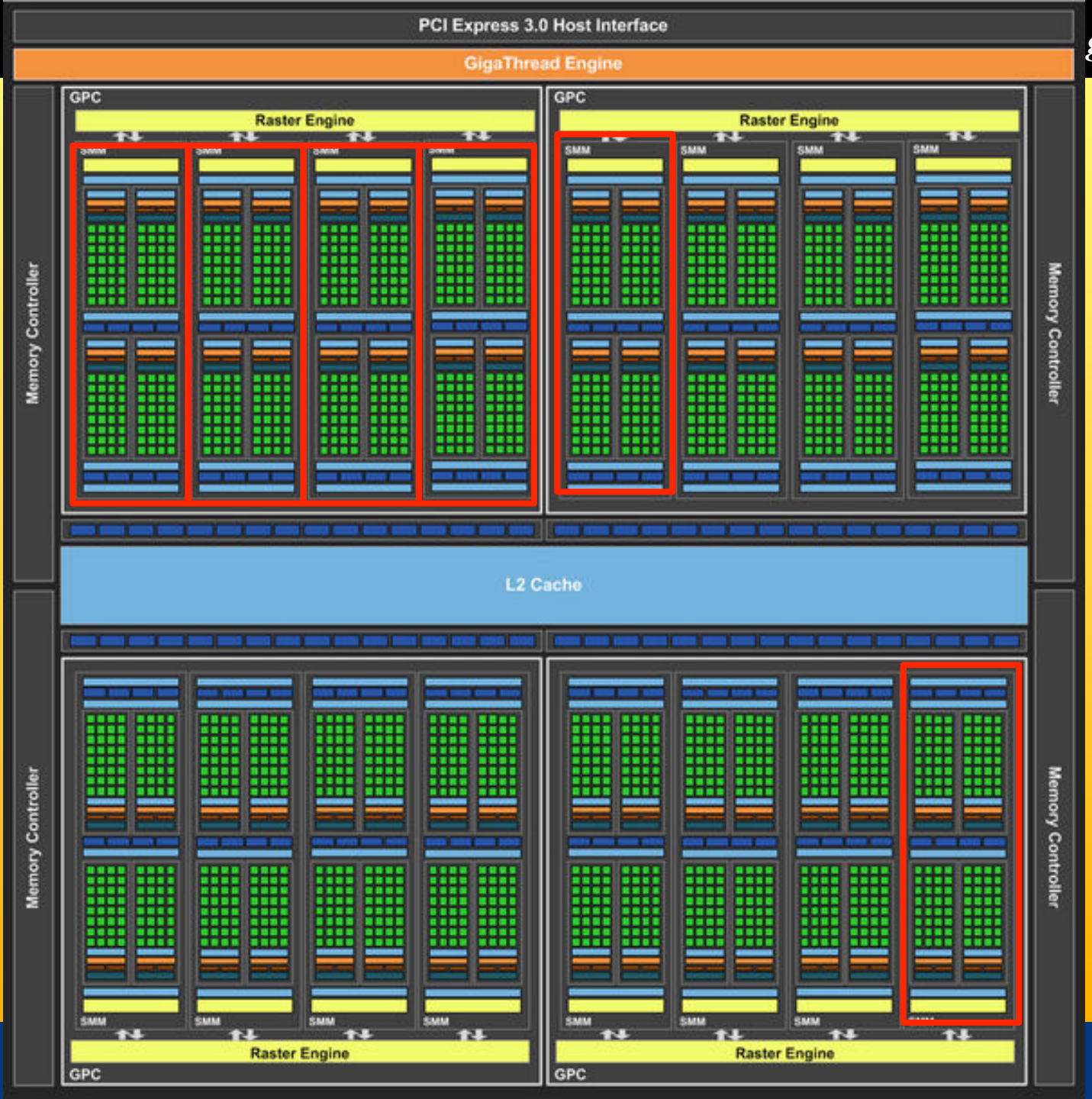
- 128 float ALUs
- 96KB L1 cache
- 8 TexUnits
- 32 Load/Store units for access to global memory



16 SMMs (“Cores”)
2MB L2 cache
64 color outputs/ clock (i.e., 64 ROPs)

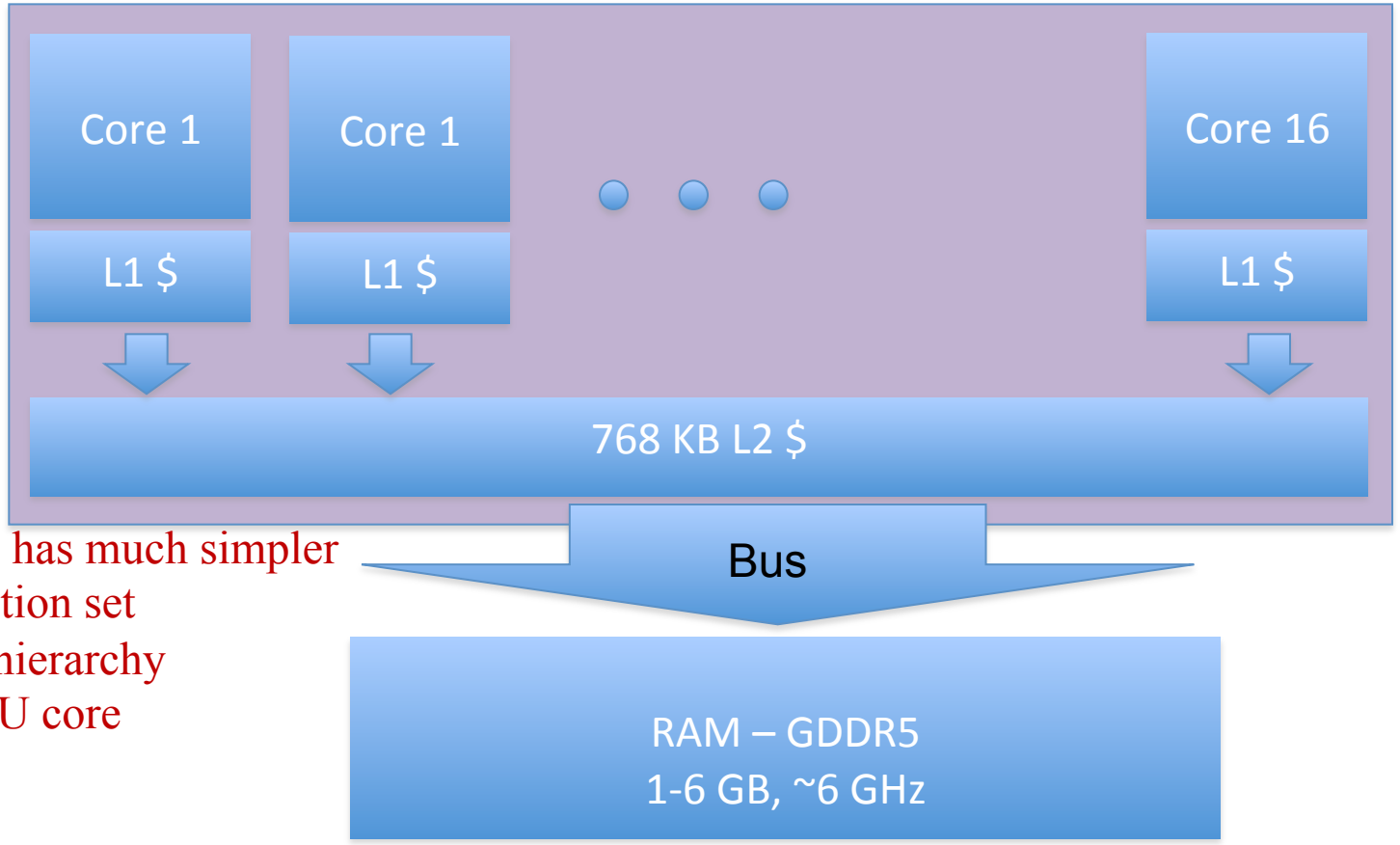
Each SMM:

- 128 ALUs
- 96KB L1 cache
- 8 TexUnits
- 32 Load/Store units for access to global memory



GPU- Nvidia's Kepler 2012

Overview:



16 cores à 192-SIMD width (2*6*16)

16/48 KB per each 48 SIMD

GPU core has much simpler
- instruction set
- cache hierarchy

than a CPU core

Bandwidth ~330 GB/s

Bus: 256/384 bits

Compare to ATI 2900:
- 2x512bits

Larrabee:
- 2x512bits

Wish:

3072 ALUs à 1 float/clock => 12KB/clock

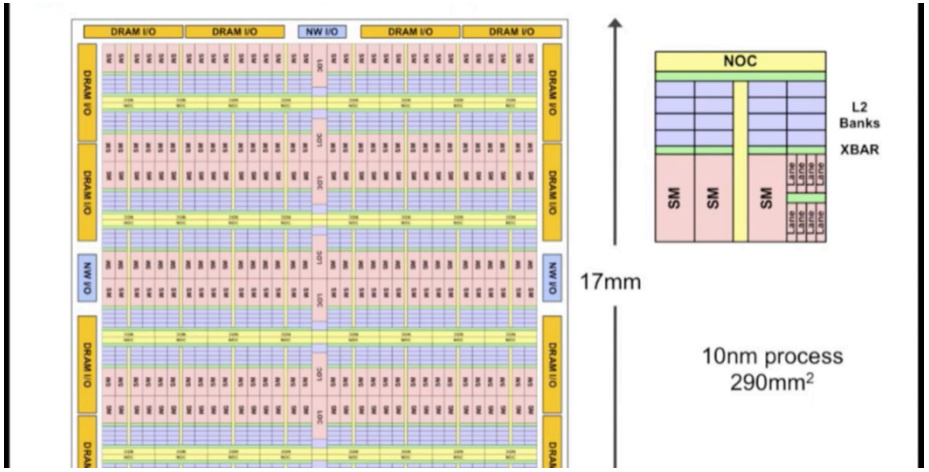
~1GHz core clock => 12000 GB/s request

We have ~330GB/s. In reality we can do 20-40 instr. between each RAM–read/write. Solved by L1$ + L2$ + latency hiding (warp switching)

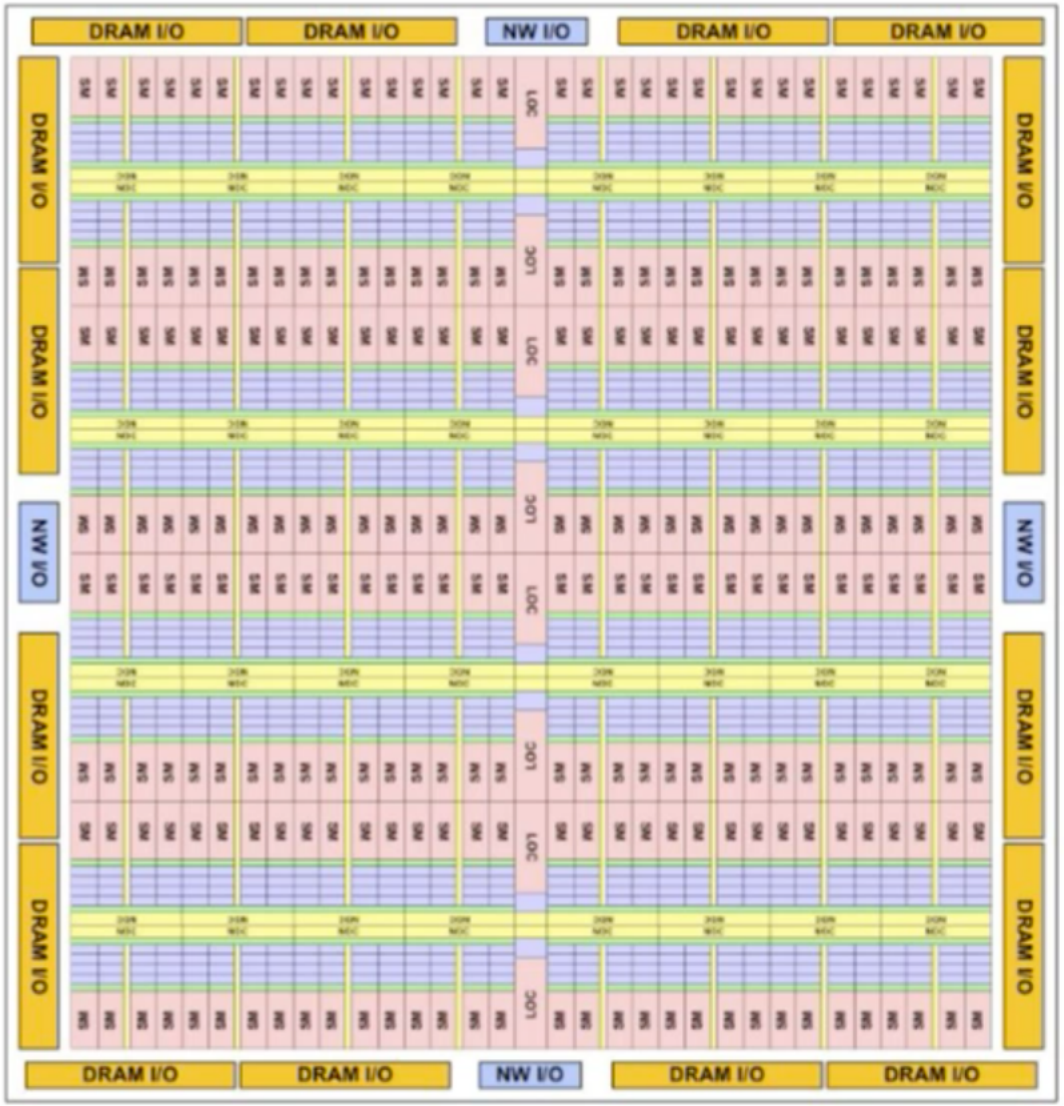
NVIDIA year 2020

- Exaflop machine:
- Google on:
  "The Challenge of Future High-Performance Computing" Uppsala
- http://media.medfarm.uu.se/play/video/3261#__utma=1.4337140.1361541635.1361541635.1361541635.1&__utmb=1.4.10.1361541635&__utmc=1&__utmx=-&__utmz=1.1361541635.1.1.utmcsr=(direct)%7Cutmccn=(direct)%7Cutmcmd=(none)&__utmv=-&__utmk=104508928
- Bill Dally, Chief Scientist & sr VP of Research, NVIDIA, prof. of Engineering, Stanford Univ.

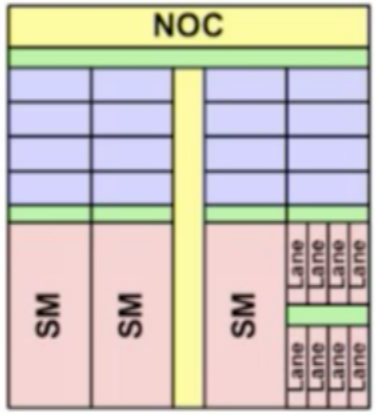
- "Energy efficiency is key to performance"
  – Flops/W.



Echelon Chip Floorplan



17mm



10nm process
290mm²

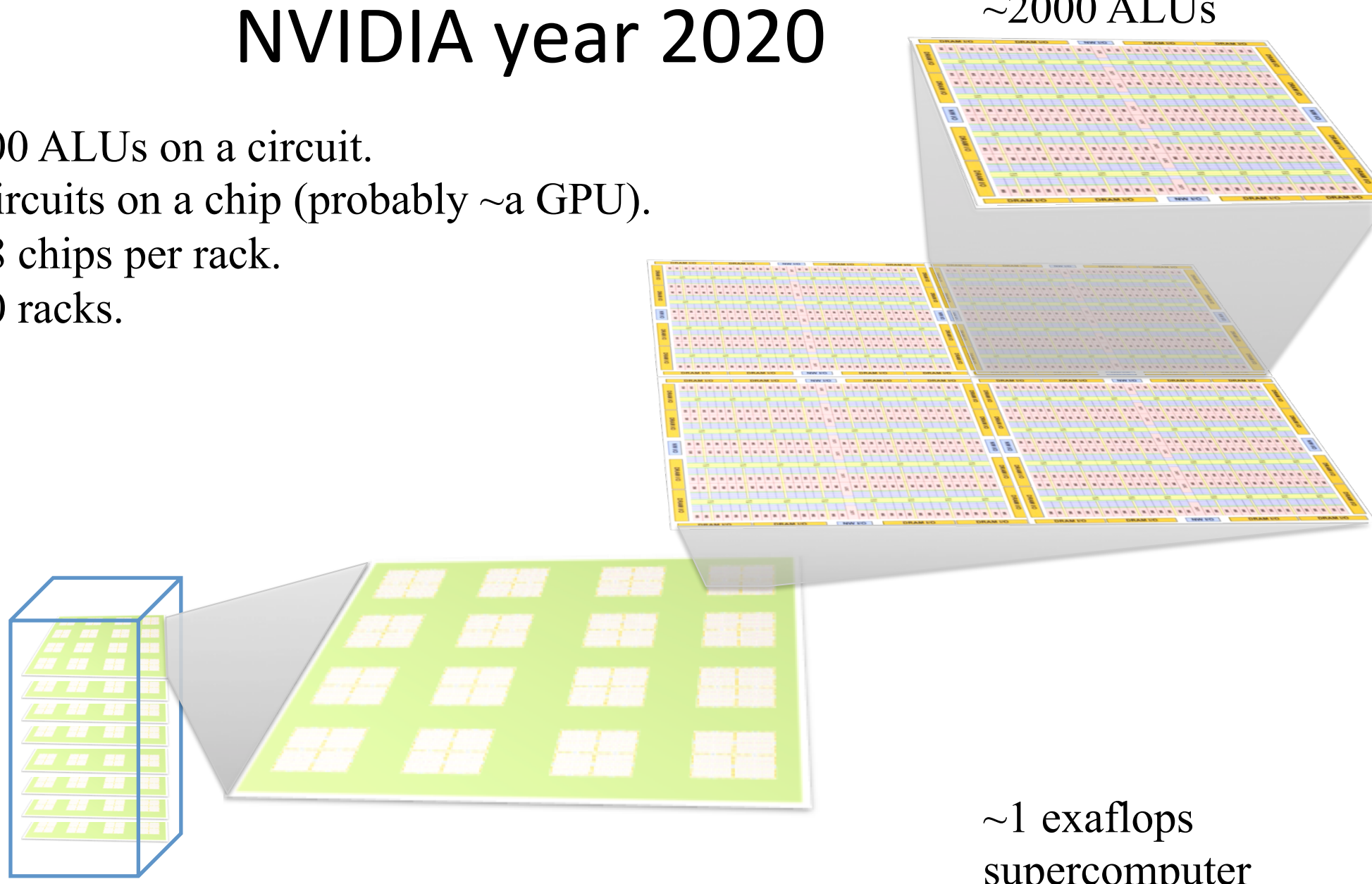
256 SMs à 8 ALUs
= 2000 ALUs

4-5 per physical chip

400 rack à 128 chip

NVIDIA year 2020

2000 ALUs on a circuit.
4 circuits on a chip (probably ~a GPU).
128 chips per rack.
400 racks.



~1 exaflops supercomputer

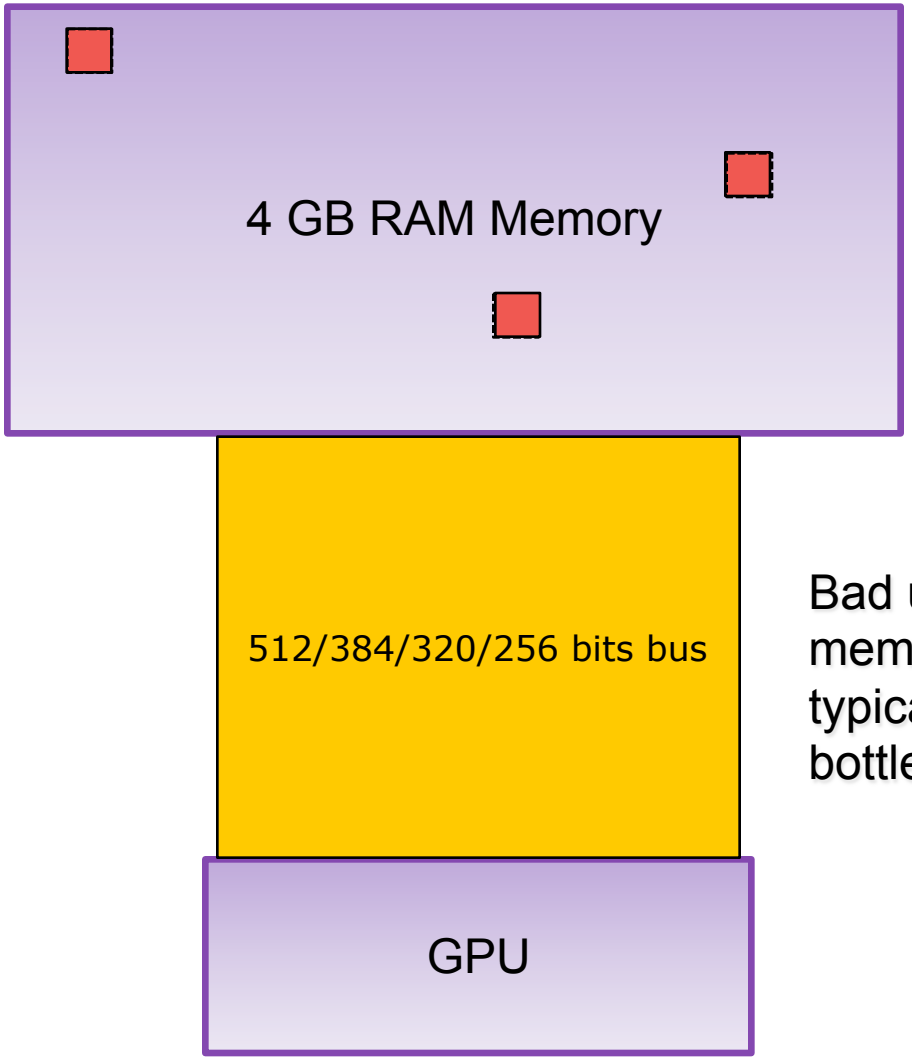
~400 racks á 128 chips

Hur skapar man optimala GPU program?

Svar: coallesced memory accesses

Graphics Processing Unit – GPU

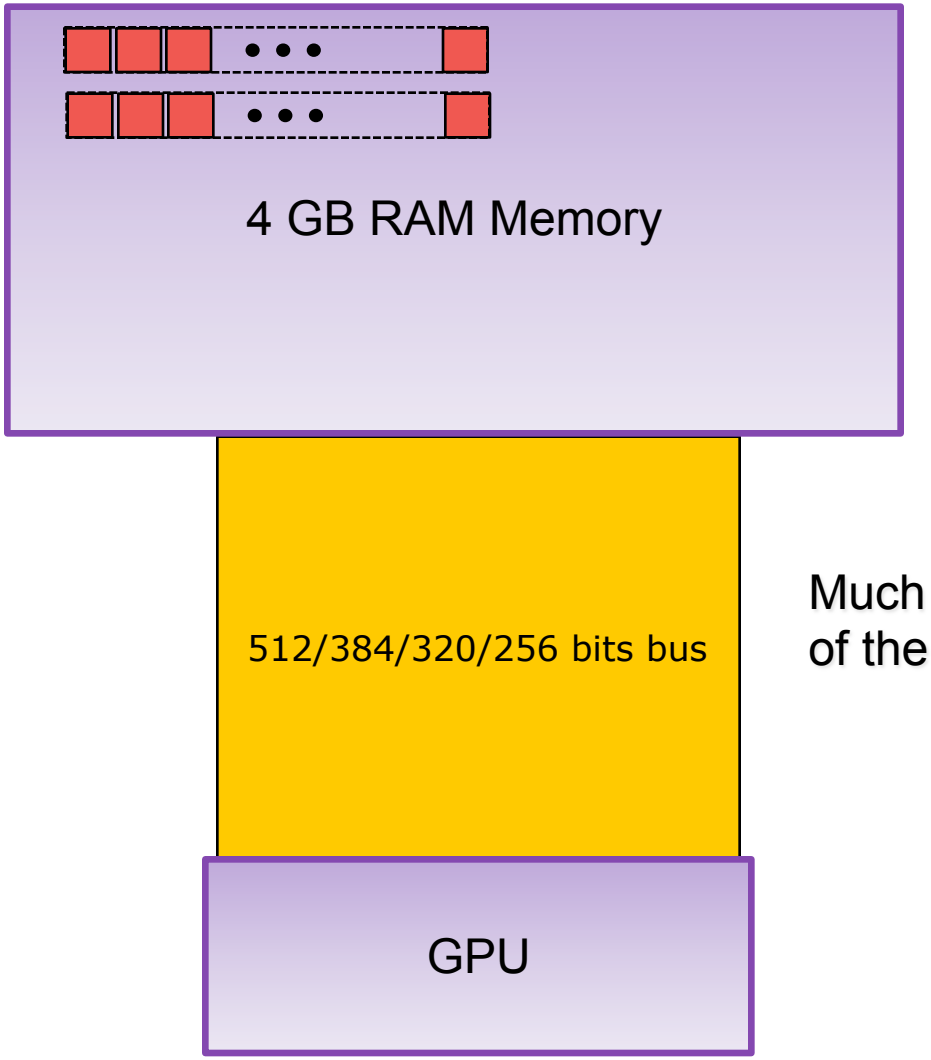
Conceptual layout:



Bad utilization of the memory bus, which typically is the bottleneck!

= memory element (32 bits)

Graphics Processing Unit - GPU

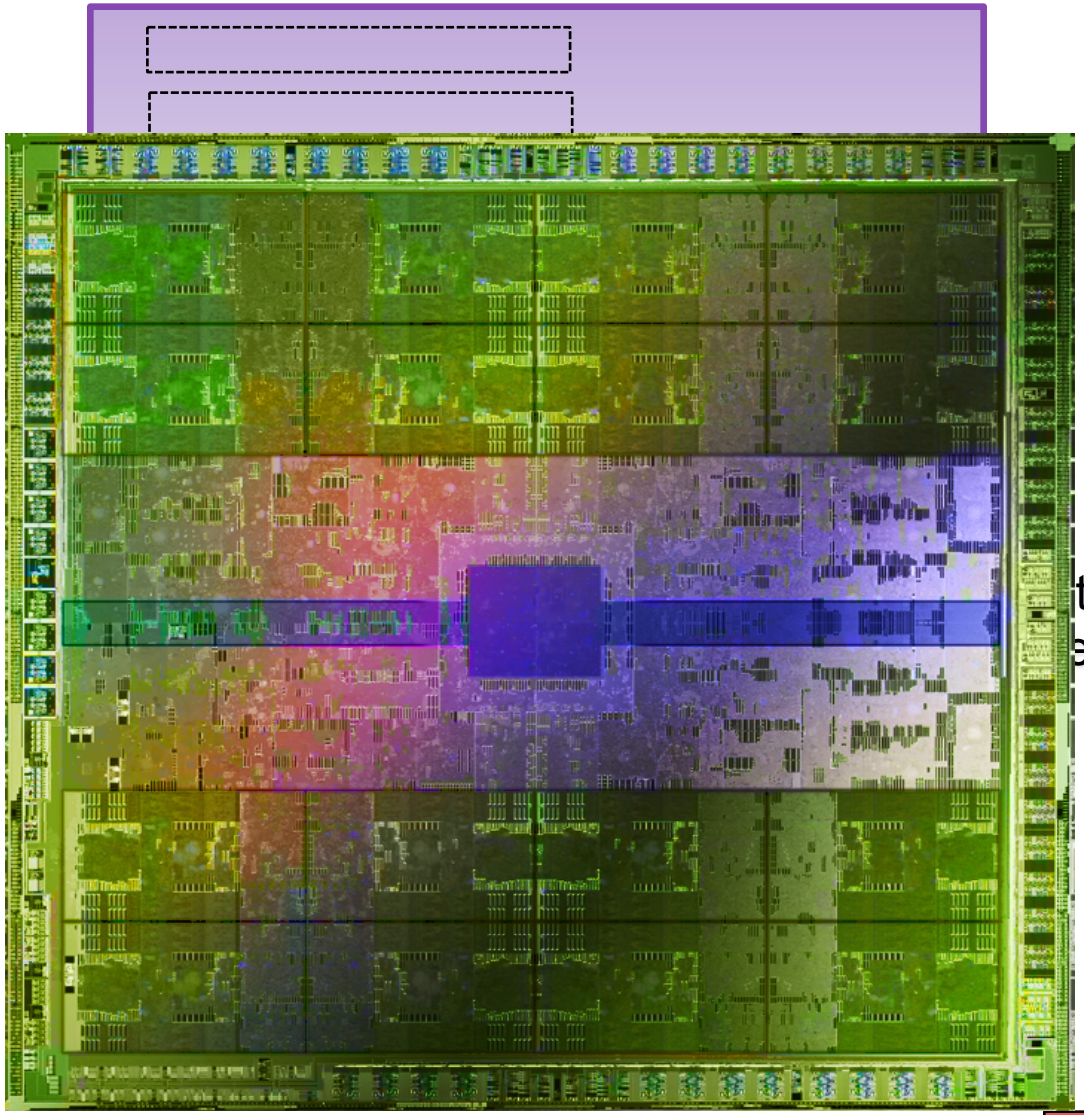


Read 32 coallesced floats for max bandwidth usage

Much better utilization of the memory bus!

= memory element (32 bits)

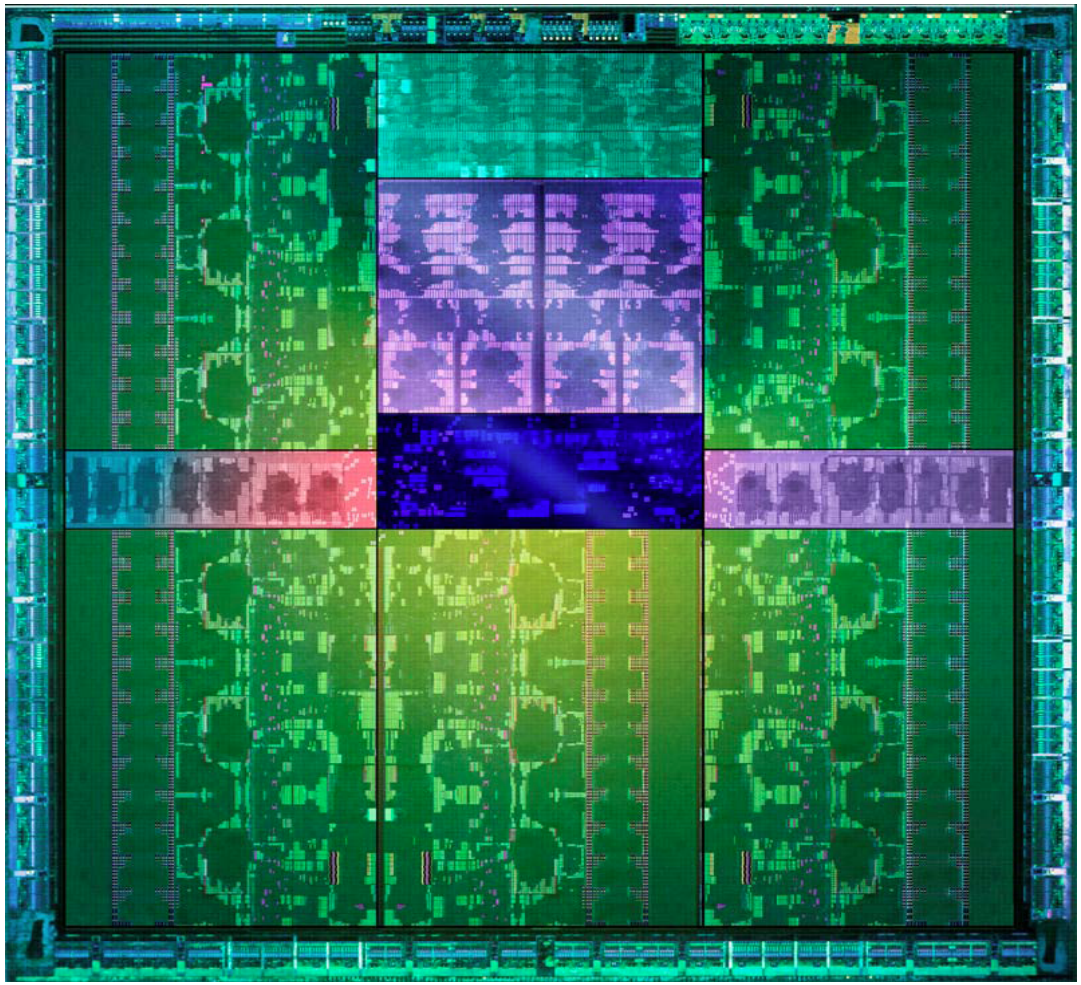
Let’s look at the GPU



tter utilization
emory bus!

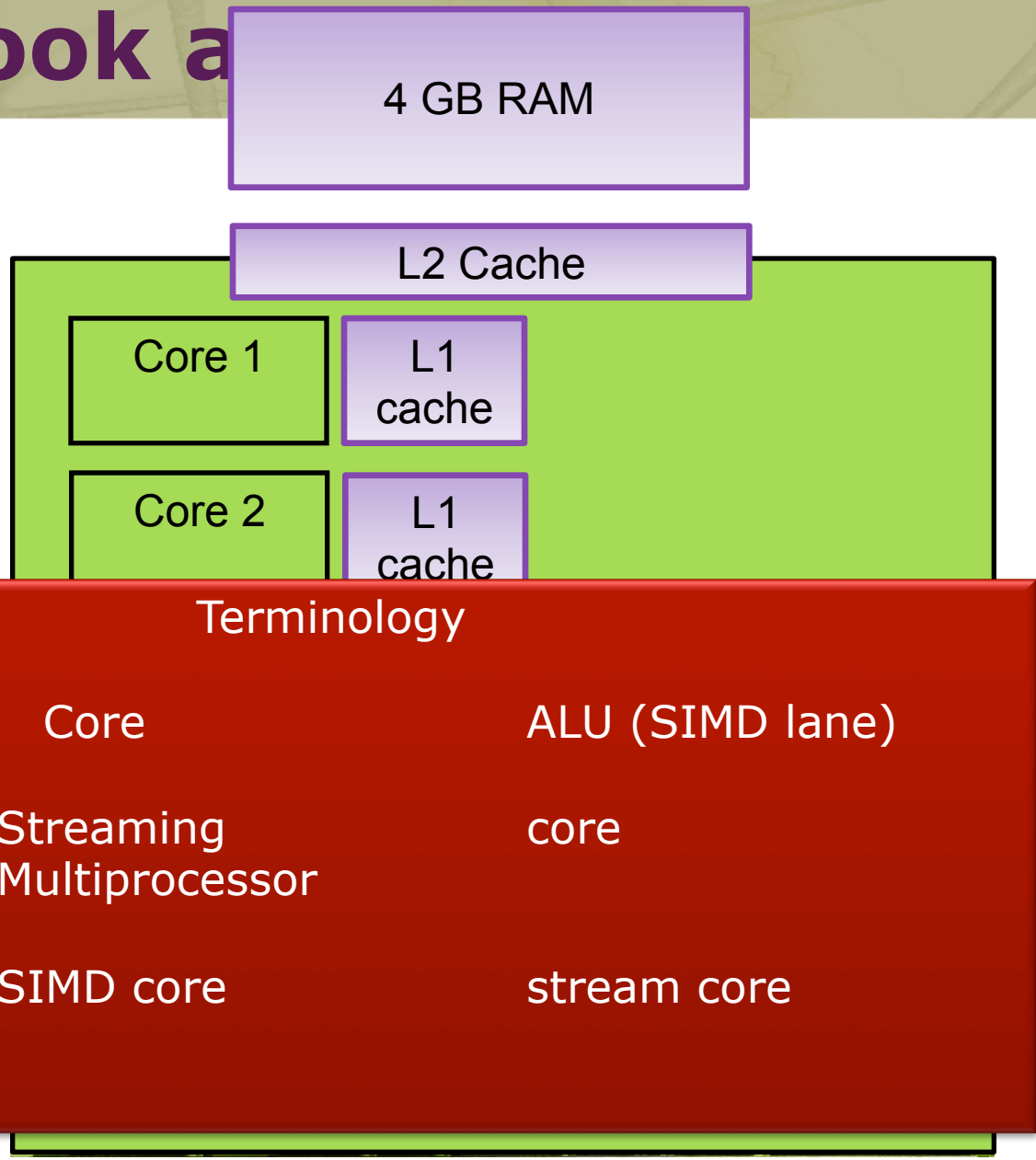
= memory element (32 bits)

Let’s look at the GPU



Kepler: 15-16 multi-processors

Let's look a



Terminology		
CPU:	Core	ALU (SIMD lane)
NVIDIA:	Streaming Multiprocessor	core
ATI	SIMD core	stream core

192 ALUs or "lanes"
(logically: 6 x 32-SIMD width)
6x32 mul/add per 1-2 clocks
(6x32 "threads")

SIMD = single instruction multiple data

Kepler: 15-16 multi-processors

Let's look at the GPU

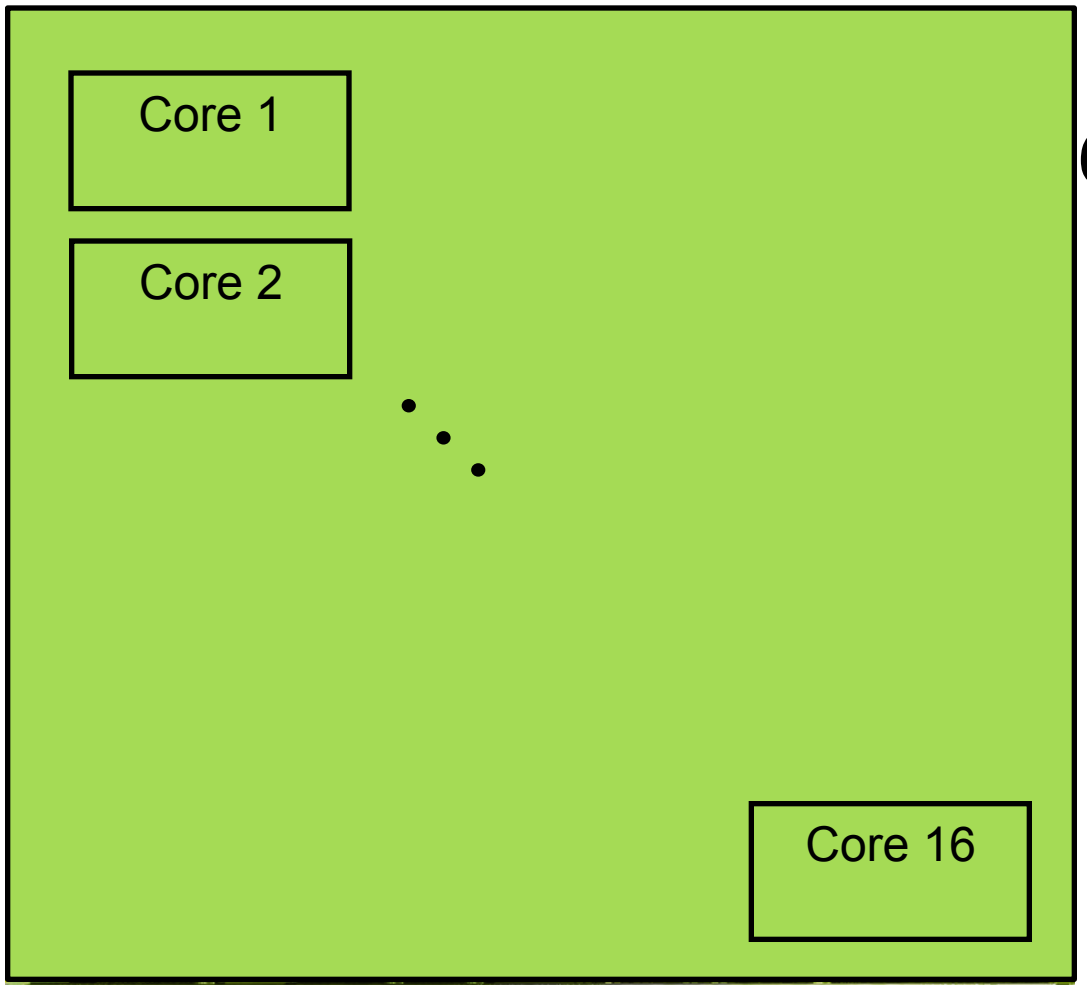
Each core:

- executes one program (=shader).

Each cycle:

- 192 flops
- 6x32 SIMD

for up to 4 different instr.



6x

From RAM or L1/L2 cache

32 add/mul etc in 2 clock cycles

To RAM or L1/L2 cache

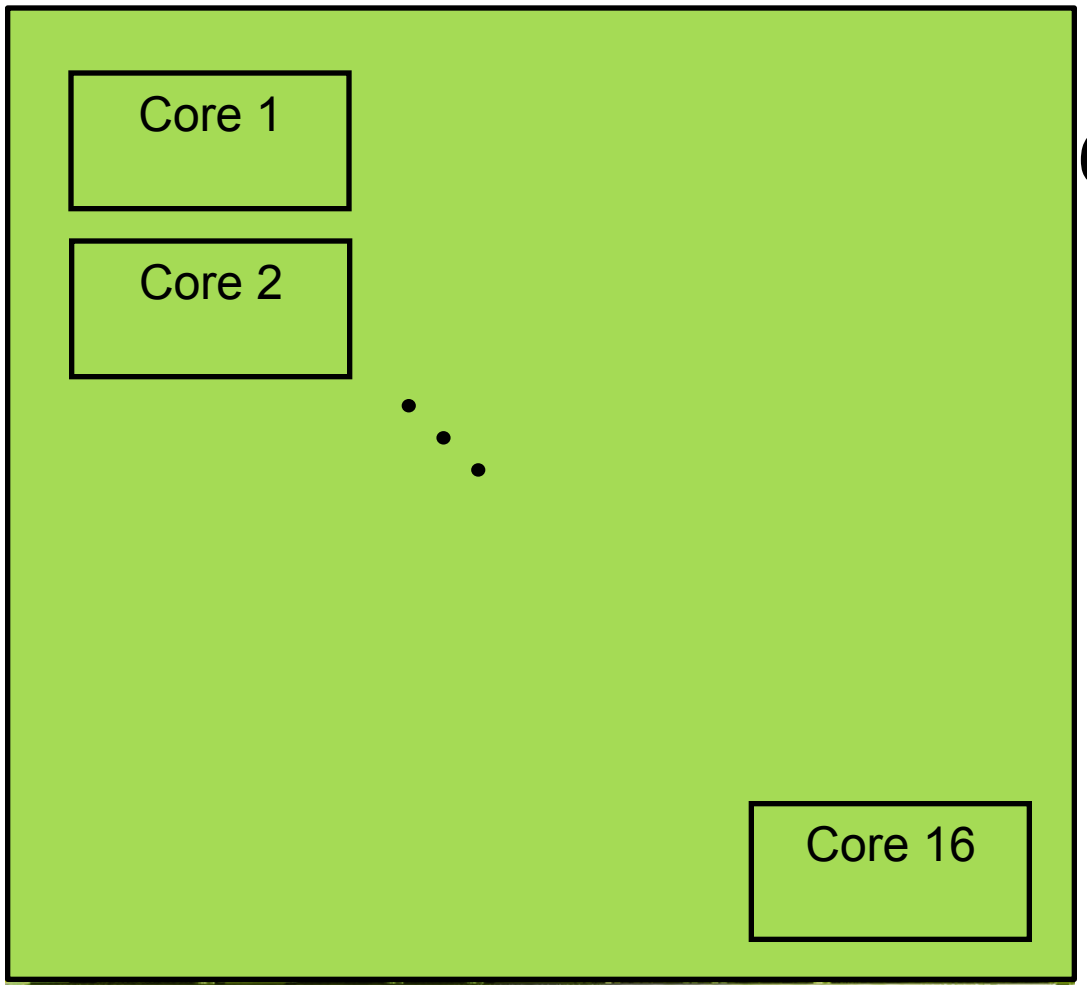
Kepler: 15-16 multi-processors

Let's look at the GPU

Each core:
- executes one program (=shader).

Each cycle:
- 192 flops
- 6x32 SIMD

for up to 4 different instr.



6x

From RAM or L1/L2 cache

32 add/mul etc in 2 clock cycles

To RAM or L1/L2 cache

Kepler: 15-16 multi-processors

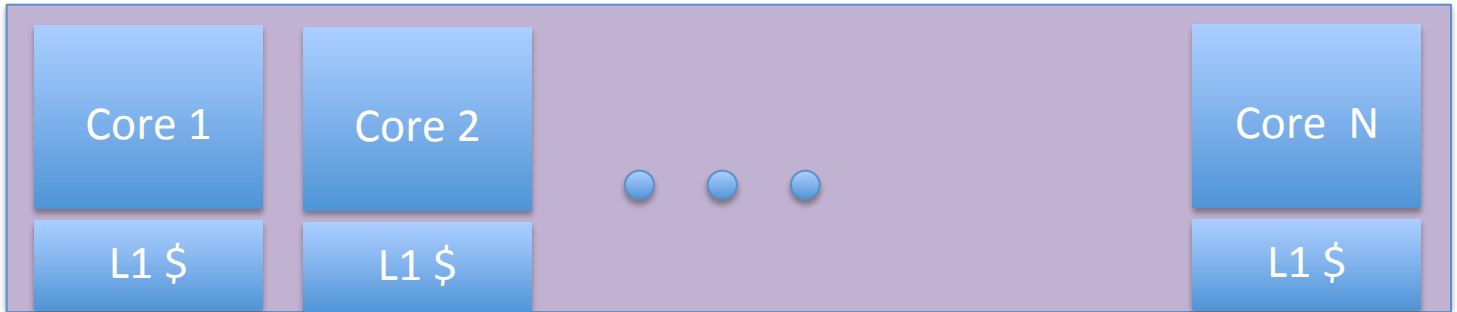
Low level APIs for GPU programming

- CUDA
  - C++ compiler
  - Works best for NVIDIA GPUs
  - CUDA SDK
    - Numerous examples and documentation (most for single GPU)
    - Has most functionality
- OpenCL
  - C compiler
  - Platform independent
    - AMD
    - NVIDIA
  - Less control/functionality than CUDA
- Compute Shaders, was Direct3D, Windows. Now available in OpenGL.
  - Upcoming. Still new.

CUDA

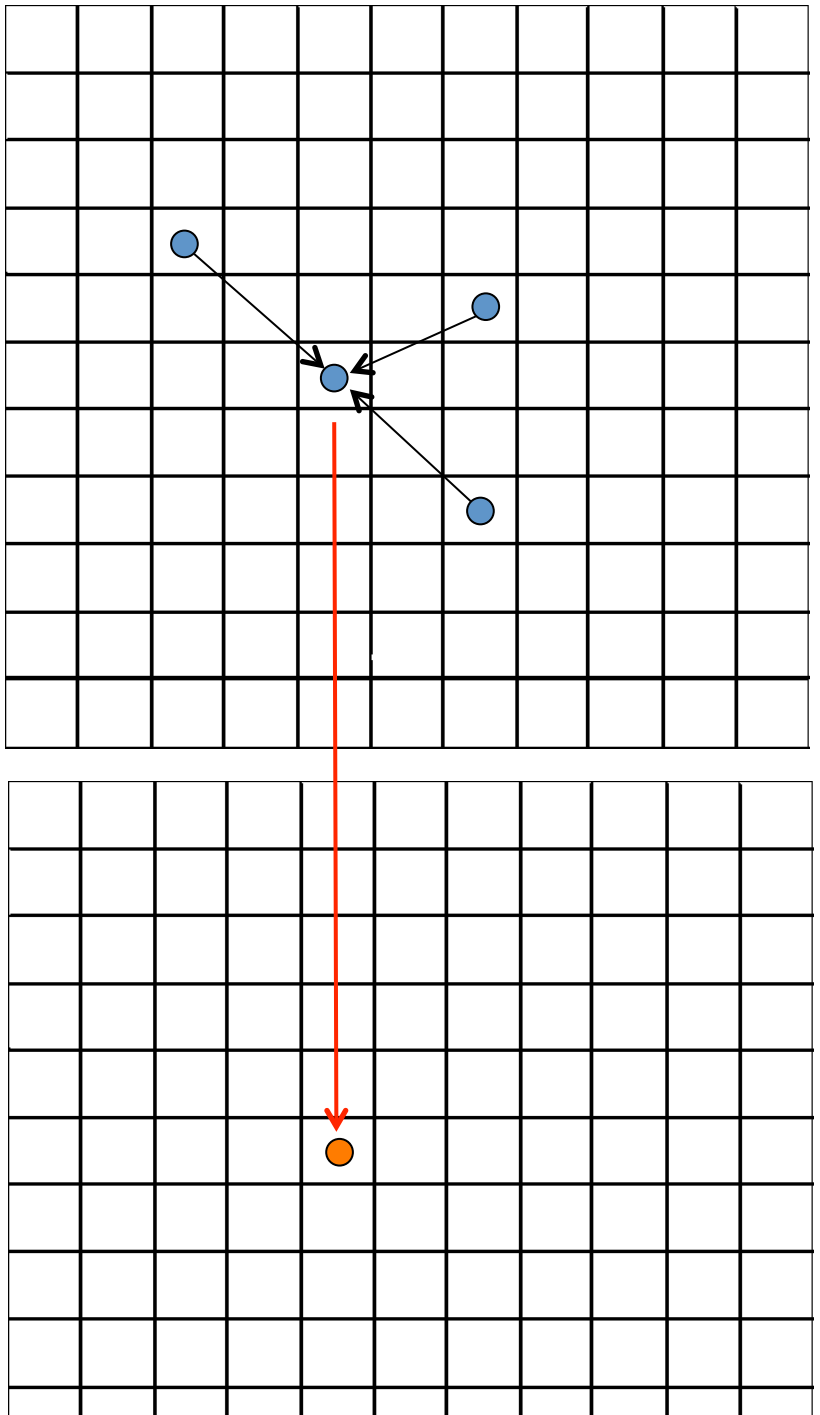
- A kernel (=CUDA program) is executed by 100:s-1M:s threads. One thread per ALU
  - A ”warp” = 32 threads. 6 warps to fill a core (=SMM = 196 ALUs)
  - Warps (one to ~32) are grouped into one block
  - Block: executed on one core
    - One to 48 warps execute on a core

Core: runs a program (kernel)
Program: [Blocks] [Threads]



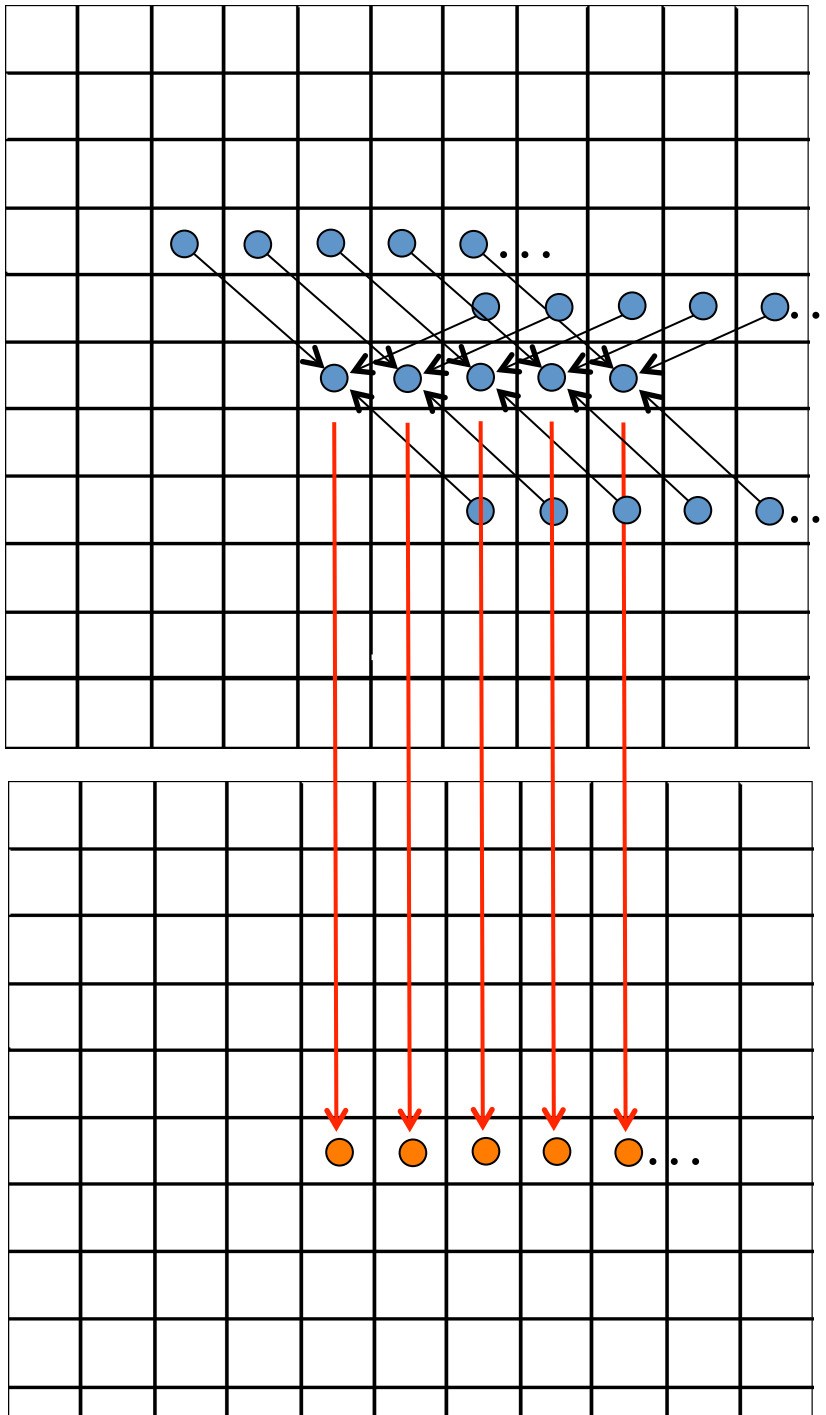
Typical Usage

- One thread per output element
  - Matrix problems
  - Grid problem
  - E.g., 1000^2, 1000^3…
- All threads execute the same program
- But each thread knows block and thread ID
  - So we can branch on this
  - Efficiency issues



Typical Usage

- One thread per input element
  - Matrix problems
  - Grid problem
  - E.g., 1000^2, 1000^3…
- All threads execute the same program
- But each thread knows block and thread ID
  - So we can branch on this
  - Efficiency issues

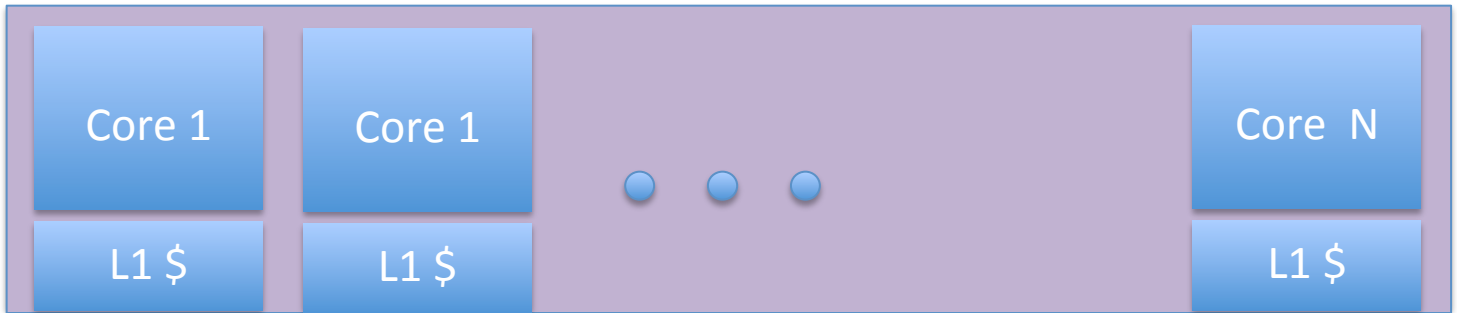


Thread utilization

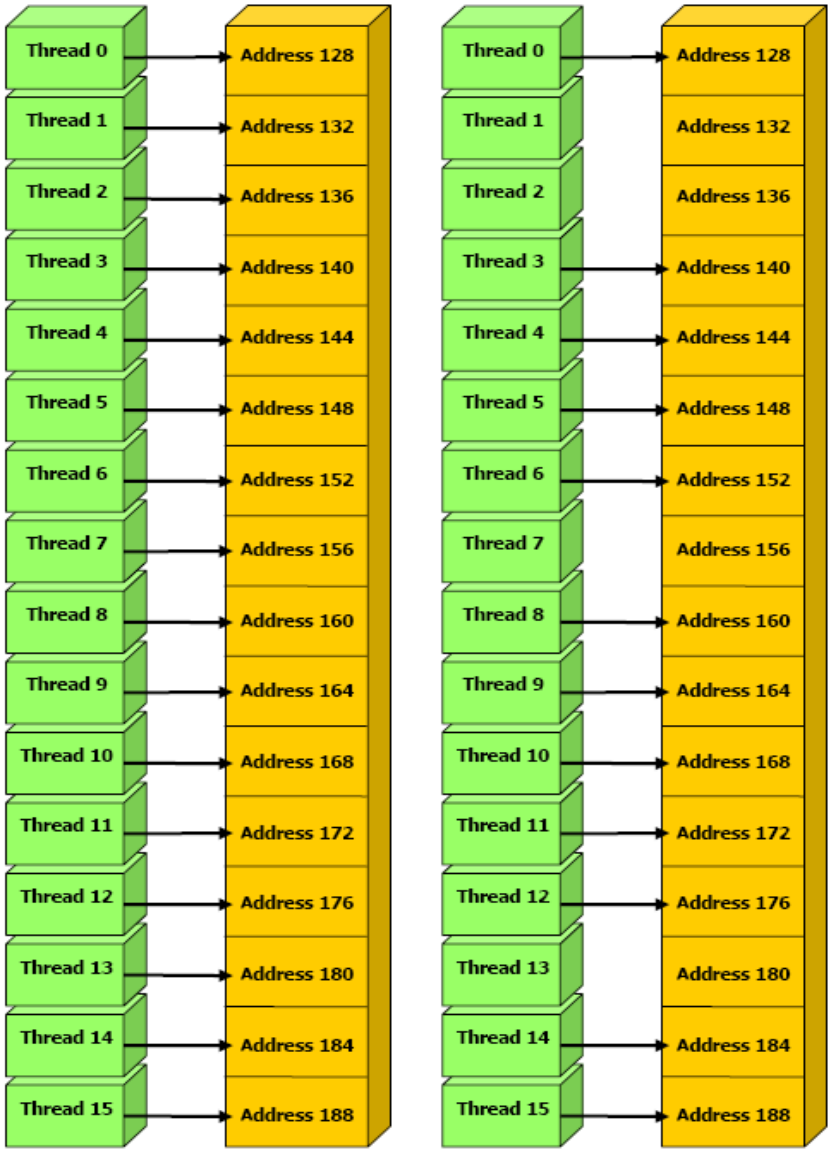
- If (…)
  - Then, a = b + c;
  - …
- Else
  - a = c + d;

The core must execute both paths if any of the 32 threads need the if and else-path.

But not if all need the same path.



Memory Acceses – Global Memory



4 GB RAM

- Coalesced reads and writes
- For maximum performance, each thread should read from the same 16-float block (128 bytes)
  - i.e., the same cache-line

Fermi

- Global mem accesses.
- One transaction:

Bandwidth to GPU RAM is the most precious resource, so two transactions is often bad.

- Two transactions:

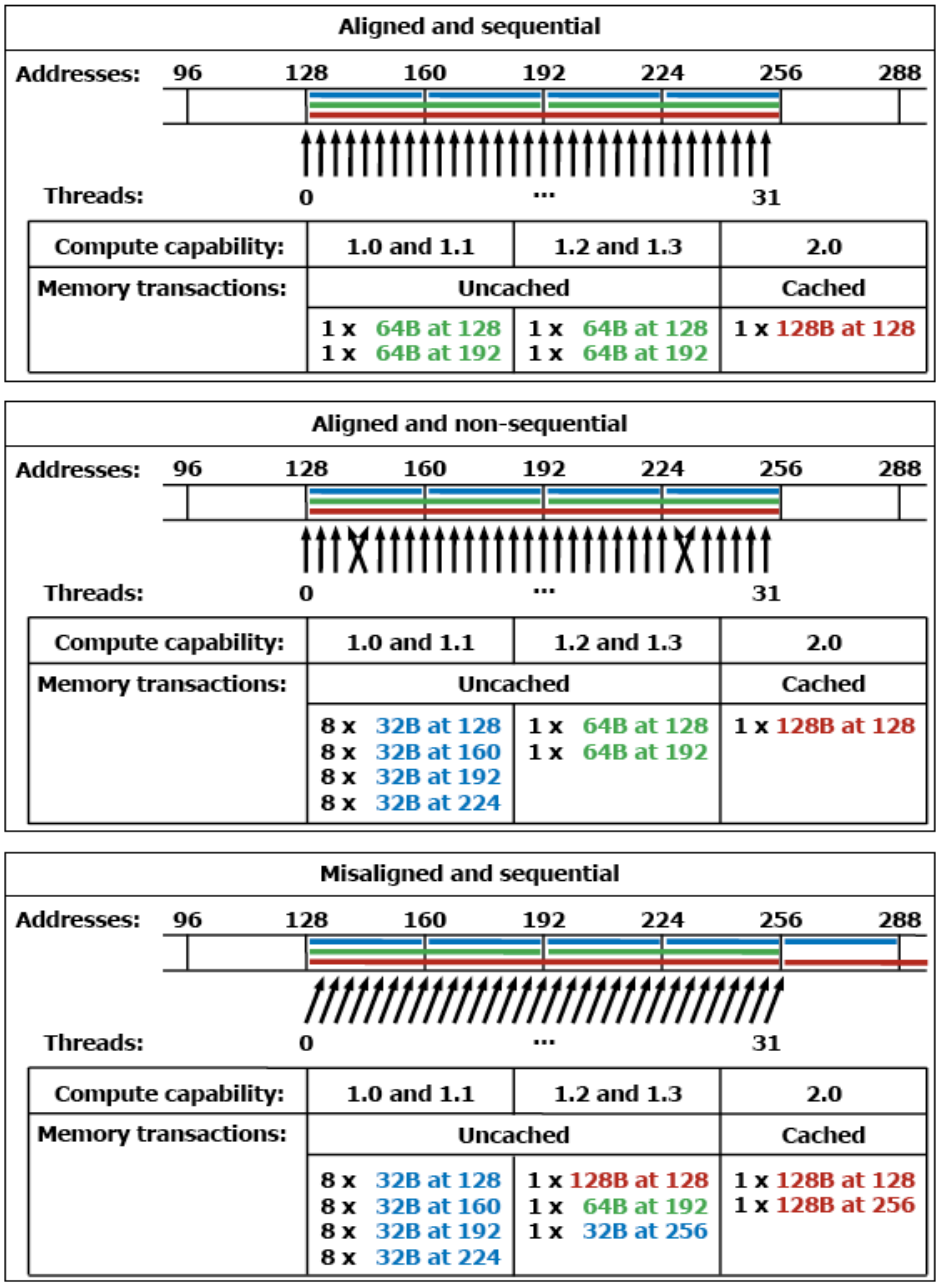


Figure G-1. Examples of Global Memory Accesses by a Warp, 4-Byte Word per Thread, and Associated Memory Transactions Based on Compute Capability

Anatomy of a CUDA Program

- "Host" program
  - runs on the CPU
  - perhaps written in Python or C++

- "Kernel"
  - code that runs on the GPU
  - does the heavy lifting
  - written in CUDA (or OpenCL)
- Your .cpp file needs to be compiled with NVIDIA's nvcc compiler.

Anatomy of a CUDA Program

Host **GPU**

1. Initialize GPU
   - transfer data to GPU
   - …

Anatomy of a CUDA Program

Host **GPU**

1. Initialize GPU
   - transfer data to GPU
   - …
2. Launch Kernel

Anatomy of a CUDA Program

Host

1. Initialize GPU
   - transfer data to GPU
   - …
2. Launch Kernel
3. Wait
   - or better: do something else
   - queue other kernels?

GPU

- Execute kernel

Anatomy of a CUDA Program

Host

1. Initialize GPU
   - transfer data to GPU
   - …
2. Launch Kernel
3. Wait
   - or better: do something else
   - queue other kernels?

GPU

- Execute kernel
- Done
- Execute next kernel

Anatomy of a CUDA Program

Host

1. Initialize GPU
   - transfer data to GPU
   - …
2. Launch Kernel
3. Wait
   - or better: do something else
   - queue other kernels?
4. Get data from GPU

GPU

- Execute kernel
- Done
- Execute next kernel

Theory to Practice…

- "Hello, World!" – GPU-style
  - as tradition dictates…

Recap: Hello, World

Python: hello_world.py

```
print( “Hello, world!” );
```

C/C++: hello_world.cpp

```
#include <cstdio>

int main()
{
   printf( “Hello, world\n” );
   return 0;
}
```

Recap: Hello, World

Python -- Result

```
$ python ./hello_world.py
Hello, world!
$
```

C/C++ -- Result

```
$ g++ hello_world.cpp
$ ./a.out
Hello, world!
$
```

Hello, World – in CUDA

```
#include <cstdio>

__global__ void hello_kernel()
{
    printf( “(%d,%d) Hello, world!\n”,
       blockIdx.x, threadIdx.x
    );
}

int main()
{
   hello_kernel<<<512,512>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

Hello, World – in CUDA

```
#include <cstdio>

__global__ void hello_kernel()
{
    printf( "(%d,%d) Hello, world!\n",
       blockIdx.x, threadIdx.x
    );
}

int main()
{
   hello_kernel<<<512,512>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

Kernel

Hello, World – in CUDA

```
#include <cstdio>

__global__ void hello_kernel()
{
    printf( "(%d,%d) Hello, world!\n",
       blockIdx.x, threadIdx.x
    );
}

int main()
{
   hello_kernel<<<512,512>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

Kernel

Hello, World – in CUDA

```
#include <cstdio>

__global__ void hello_kernel()
{
    printf( "(%d,%d) Hello, world!\n",
        blockIdx.x, threadIdx.x
    );
}

int mai
{
    hell
    cuda
    retu
}
```

Prints "Hello, world!", prefixed by two numbers identifying the thread. Example:

(0,0) Hello, world!

Hello, World – in CUDA

```
#include <cstdio>

__global__ void hello_kernel()
{
    printf( "(%d,%d) Hello, world!\n",
        blockIdx.x, threadIdx.x
    );
}

int main()
{
   hello_kernel
   cudaDeviceSyn
   return 0;
}
```

Special variables, provided by CUDA.
Identify the current thread.

Hello, World – in CUDA

```
#include <cstdio>

__global__ void hello_kernel()
{
    printf( “(%d,%d) Hello, world!\n”,
       blockIdx.x, threadIdx.x
    );
}
```

```
int main()
{
   hello_kernel<<<512,512>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

Host

Hello, W

```
#include <cstd

__global__ voi
{
    printf( "(
        blockId
    );
}

int main()
{
    hello_kernel<<<512,512>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Launch the kernel "hello_kernel".

Run it in 512 blocks with 512 threads each.

So, in total, run the kernel function 512×512 = 262144 times…

Hello, World – in CUDA

```
#include <cstdio>

__global__ void hello
{
    printf( "(%d,%d) H
       blockIdx.x, thr
    );
}

int main()
{
   hello_kernel<<<512,512>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

Wait for GPU to finish.

Hello, World – in CUDA – Result

```
$ nvcc –arch compute_20 hello_world.cu
$ ./a.out
(28,480) Hello, world!
(28,481) Hello, world!
(28,496) Hello, world!
(28,482) Hello, world!
(28,483) Hello, world!
(28,497) Hello, world!
(28,484) Hello, world!
(28,485) Hello, world!
```

Hello, World – in CUDA – Result, 2

(28,486) Hello, world!
(28,498) Hello, world!
(28,487) Hello, world!
(28,499) Hello, world!
(28,488) Hello, world!
(28,500) Hello, world!
(28,489) Hello, world!
(28,501) Hello, world!
(28,490) Hello, world!
(28,502) Hello, world!
(28,491) Hello, world!
(28,503) Hello, world!
(28,492) Hello, world!
(28,493) Hello, world!
(28,494) Hello, world!
(28,495) Hello, world!
(28,504) Hello, world!
(28,505) Hello, world!
(28,506) Hello, world!
(28,507) Hello, world!
(28,508) Hello, world!
(28,509) Hello, world!
(28,510) Hello, world!
(28,511) Hello, world!
(12,448) Hello, world!
(12,464) Hello, world!
(12,449) Hello, world!
(12,450) Hello, world!
(12,465) Hello, world!
(12,451) Hello, world!
(12,452) Hello, world!
(12,466) Hello, world!
(12,453) Hello, world!
(12,454) Hello, world!

Hello, World – in CUDA – Result, 3

(12,467) Hello, world!
(12,455) Hello, world!
(12,468) Hello, world!
(12,456) Hello, world!
(12,469) Hello, world!
(12,457) Hello, world!
(12,470) Hello, world!
(12,458) Hello, world!
(12,471) Hello, world!
(12,459) Hello, world!
(12,472) Hello, world!
(12,460) Hello, world!
(12,473) Hello, world!
(12,461) Hello, world!
(12,474) Hello, world!
(12,462) Hello, world!
(12,475) Hello, world!
(12,463) Hello, world!
(12,476) Hello, world!
(12,477) Hello, world!
(12,478) Hello, world!
(12,479) Hello, world!
(1,128) Hello, world!
(1,129) Hello, world!
(1,144) Hello, world!
(1,130) Hello, world!
(1,131) Hello, world!
(1,145) Hello, world!
(1,132) Hello, world!
(1,146) Hello, world!
(1,133) Hello, world!
(1,147) Hello, world!
(1,134) Hello, world!
(1,148) Hello, world!

Results, Summary

- I'll skip the remaining 7700 or so slides
  - for approximately 262000 lines of output

- Up to ~ 20000 threads executed at once
- Arbitrary execution order
  - let's look at the output again.

Ordering…

(12,467) Hello, world!
(12,455) Hello, world!
(12,468) Hello, world!
(12,456) Hello, world!
(12,469) Hello, world!
(12,457) Hello, world!
(12,470) Hello, world!
(12,458) Hello, world!
(12,471) Hello, world!
(12,459) Hello, world!
(12,472) Hello, world!
(12,460) Hello, world!
(12,473) Hello, world!
(12,461) Hello, world!
(12,474) Hello, world!
(12,462) Hello, world!
(12,475) Hello, world!

(12,463) Hello, world!
(12,476) Hello, world!
(12,477) Hello, world!
(12,478) Hello, world!
(12,479) Hello, world!
(1,128) Hello, world!
(1,129) Hello, world!
(1,144) Hello, world!
(1,130) Hello, world!
(1,131) Hello, world!
(1,145) Hello, world!
(1,132) Hello, world!
(1,146) Hello, world!
(1,133) Hello, world!
(1,147) Hello, world!
(1,134) Hello, world!
(1,148) Hello, world!

Ordering…

(12,467) Hello, world!
(12,455) Hello, world!
(12,468) Hel
(12,456) He
(12,469) He
(12,457) He
(12,470) He
(12,458) He
(12,471) He
(12,459) He
(12,472) He
(12,460) Hello, world!
(12,473) Hello, world!
(12,461) Hello, world!
(12,474) Hello, world!
(12,462) Hello, world!
(12,475) Hello, world!

Block 12 executed before block 1!

(12,463) Hello, world!
(12,476) Hello, world!
(12,477) Hello, world!
(12,478) Hello, world!
(12,479) Hello, world!
(1,128) Hello, world!
(1,129) Hello, world!
(1,144) Hello, world!
(1,130) Hello, world!
(1,131) Hello, world!
(1,145) Hello, world!
(1,132) Hello, world!
(1,146) Hello, world!
(1,133) Hello, world!
(1,147) Hello, world!
(1,134) Hello, world!
(1,148) Hello, world!

Ordering…

(12,467) Hello, world!
(12,455) Hello, world!
(12,468) Hello, world!
(12,456) Hello, world!
(12,469) Hello, world!
(12,457) Hello, world!
(12,470) Hello, world!
(12,458) Hello, world!
(12,471) Hello, world!
(12,459) Hello, world!
(12,472) Hello, world!
(12,460) Hello, world!
(12,473) Hello, world!
(12,461) Hello, world!
(12,474) Hello, world!
(12,462) Hello, world!
(12,475) Hello, world!

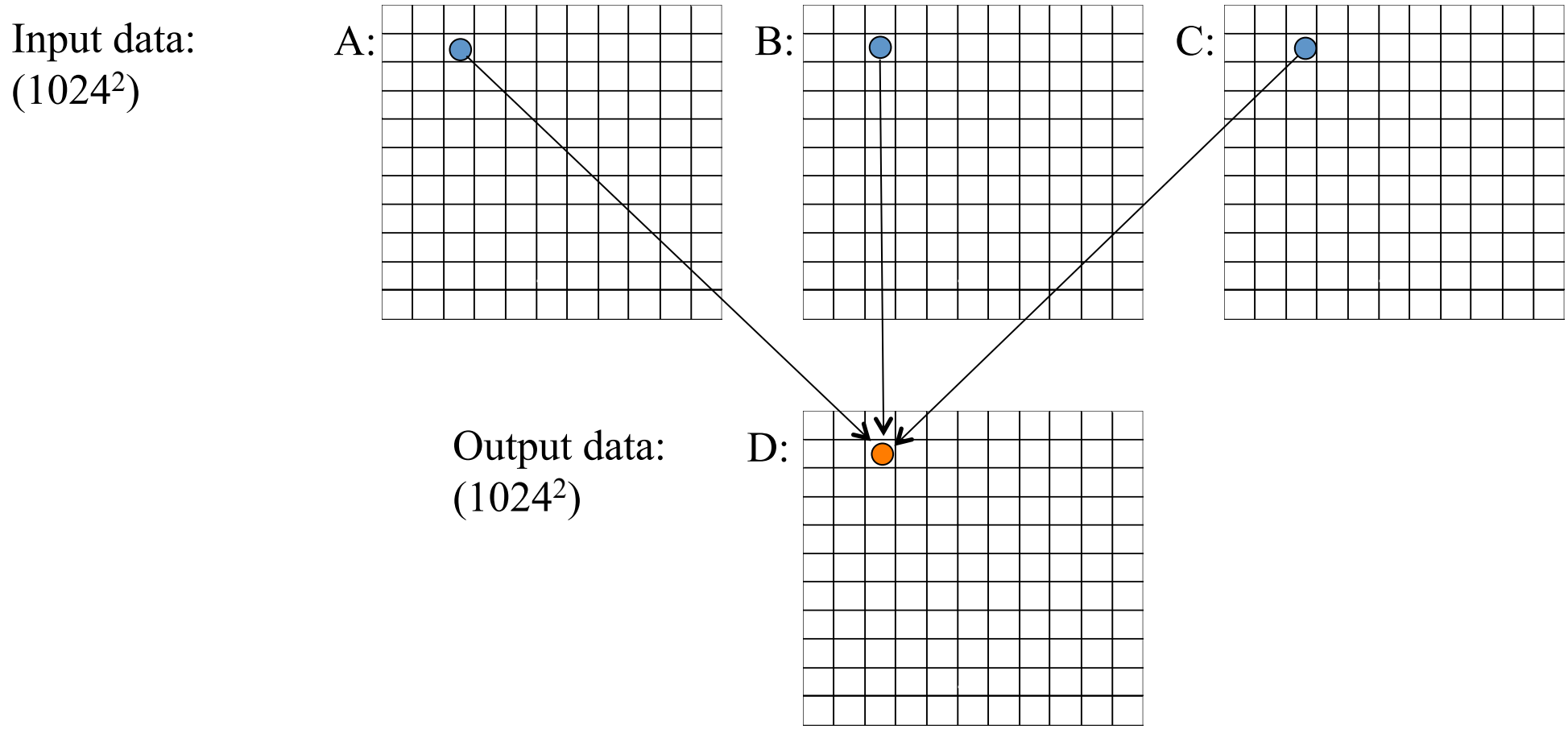
(12,463) Hello, world!

(1,148) Hello, world!

Thread 473 of Block 12 executed before Thread 461 of the same block!

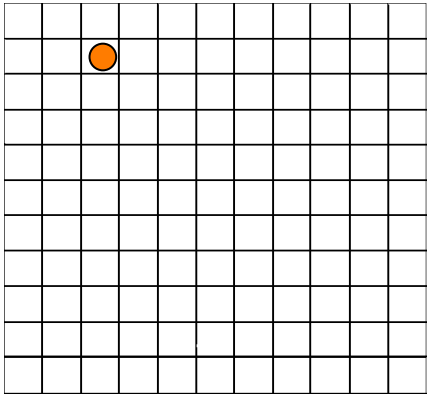
Bonus – a*b + c – Kernel

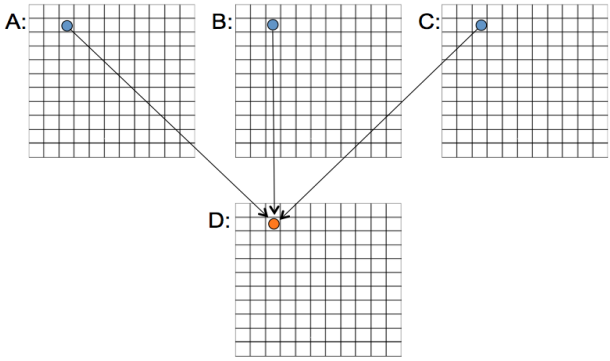
- One thread per output element
  - Result per cell: D[x,y] = A[x,y] * B[x,y] + C[x,y]



Bonus – a*b + c – Kernel

```
__global__ void mad( float* d, const float* a, const
    float* b, const float* c )
{
    const size_t self = threadIdx.x +
        blockIdx.x*blockDim.x;

    d[self] = a[self] * b[self] + c[self];
}
```





Bonus – a*b + c – Host, 1

```
int main()
{
	const size_t kElements = 1024*1024;

	// allocate GPU memory
	float *gpuDest, *gpuA, *gpuB, *gpuC;

	cudaMalloc( (void**)&gpuDest, kElements * sizeof(float) );
	cudaMalloc( (void**)&gpuA, kElements * sizeof(float) );
	cudaMalloc( (void**)&gpuB, kElements * sizeof(float) );
	cudaMalloc( (void**)&gpuC, kElements * sizeof(float) );

/*Cont'd on next slide */
```

Bonus – a*b + c – Host, 2

```
// generate data and copy to GPU
float *hostA, *hostB, *hostC;

/*
      hostA = ...;
      hostB = ...;
      hostC = ...;
 */

cudaMemcpy( gpuA, hostA, kElements*sizeof(float),
      cudaMemcpyHostToDevice );
cudaMemcpy( gpuB, hostB, kElements*sizeof(float),
      cudaMemcpyHostToDevice );
cudaMemcpy( gpuC, hostC, kElements*sizeof(float),
      cudaMemcpyHostToDevice );
```

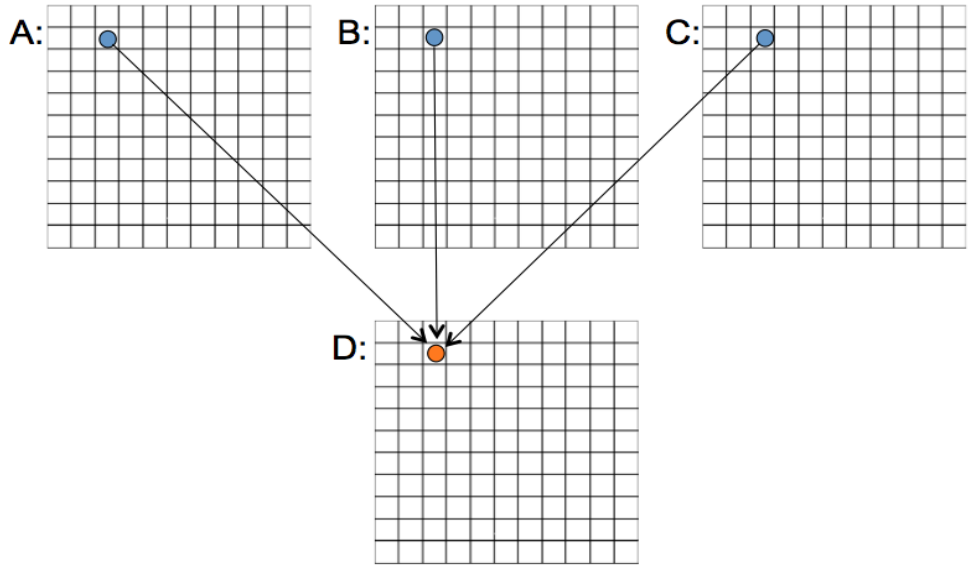
Bonus – a*b + c –

Launch the kernel “mad”.

Run it in 4096 blocks with 256 threads each.

So, in total, run the kernel function 4096×256= 1.048.576 times…

```
    // run kernel
    mad<<<1024*4,256>>>( gpuDest, gpuA, gpuB, gpuC );
    cudaDeviceSynchronize();     // Wait for GPU to finish

    // get data from GPU
    float* hostDest = new float[kElements];
    cudaMemcpy( hostDest, gpuDest,
          kElements*sizeof(float),
          cudaMemcpyDeviceToHost
     );

    /* Do something with the data! */

    return 0;
}
```



Some Libraries

- Thrust
  - open source C++ template library
  - Containers, Iterators and Algorithms.
- CUSP
  - open source C++ template library
  - sparse linear algebra and sparse matrix computations
- PETSc
  - Grids, matrices, linear/non-linear solvers and preconditioners
  - interface is provided through C, C++, FORTRAN and Python on Unix, Linux, and Windows
  - Many examples:
    - http://www.mcs.anl.gov/petsc/documentation/exercises/index.html
- Matlab
  - Need large problems to gain speedup
  - CPU->GPU transfer often a bottleneck

Thrust: Sorting 32M numbers in ~32ms

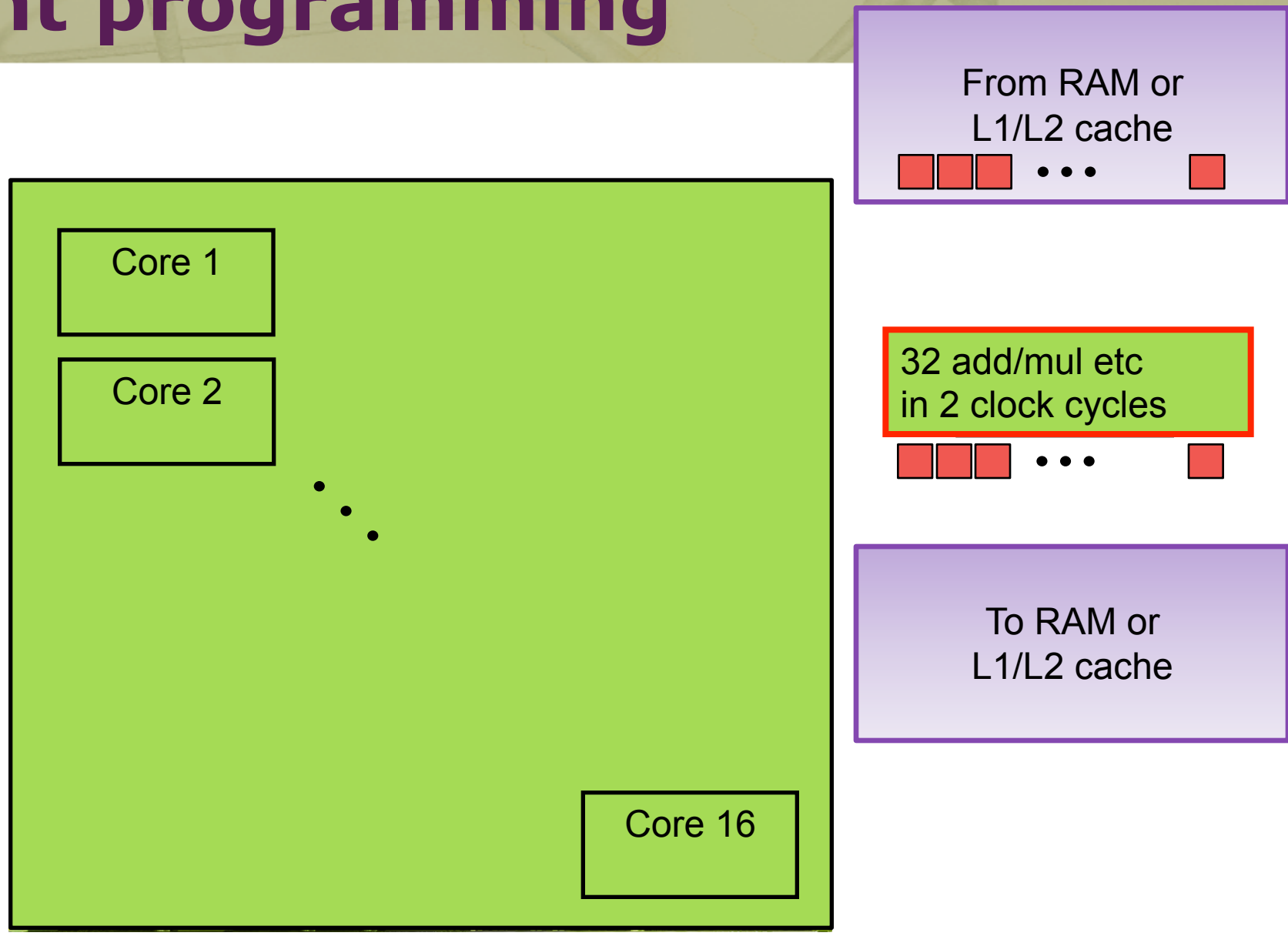
```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <algorithm>
#include <cstdlib>

int main(void)
{
  // generate 32M random numbers serially
  thrust::host_vector<int> h_vec(32 << 20);
  std::generate(h_vec.begin(), h_vec.end(), rand);

  // transfer data to the device
  thrust::device_vector<int> d_vec = h_vec;

  // sort data on the device (846M keys per second on GeForce GTX 480)
  thrust::sort(d_vec.begin(), d_vec.end());

  // transfer data back to host
  thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

  return 0;
}
```

Some Libraries

CUSP:
Solving A*x = b with the Conjugate Gradient method

```
#include <cusp/hyb_matrix.h>
#include <cusp/io/matrix_market.h>
#include <cusp/krylov/cg.h>

int main(void)
{
    // create an empty sparse matrix structure (HYB format)
    cusp::hyb_matrix<int, float, cusp::device_memory> A;

    // load a matrix stored in MatrixMarket format
    cusp::io::read_matrix_market_file(A, "5pt_10x10.mtx");

    // allocate storage for solution (x) and right hand side (b)
    cusp::array1d<float, cusp::device_memory> x(A.num_rows, 0);
    cusp::array1d<float, cusp::device_memory> b(A.num_rows, 1);

    // solve the linear system A * x = b with the Conjugate Gradient method
    cusp::krylov::cg(A, x, b);

    return 0;
}
```

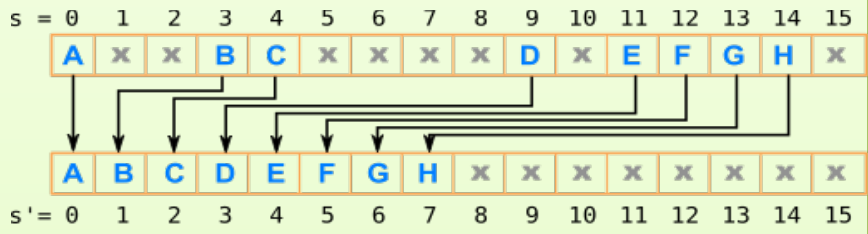
Efficient programming



Fermi: 16 multi-processors à 2x16 SIMD width

Efficient Programming

- If your program can be constructed this way, you are a winner!
- More often possible than anticipated
  - Stream compaction
  - Prefix sums
  - Sorting



19 5 100 1 63 79

1 5 19 63 79 100

input	1	3	9	4	2	5	7	1	8	4	5	9	3
output	0	1	4	13	15	…	…	…	…	…	…	…	…

Fermi: 16 multi-processors à 2x16 SIMD width

Parallelism

- Programming massively parallel systems

Parallelism

- Programming massively parallel systems
- Parallelizing algorithms

Parallelism

- Programming massively parallel systems
- Parallelizing algorithms
- Our research on 3 key components:
  1. Stream compaction
  2. Prefix Sum
  3. Sorting

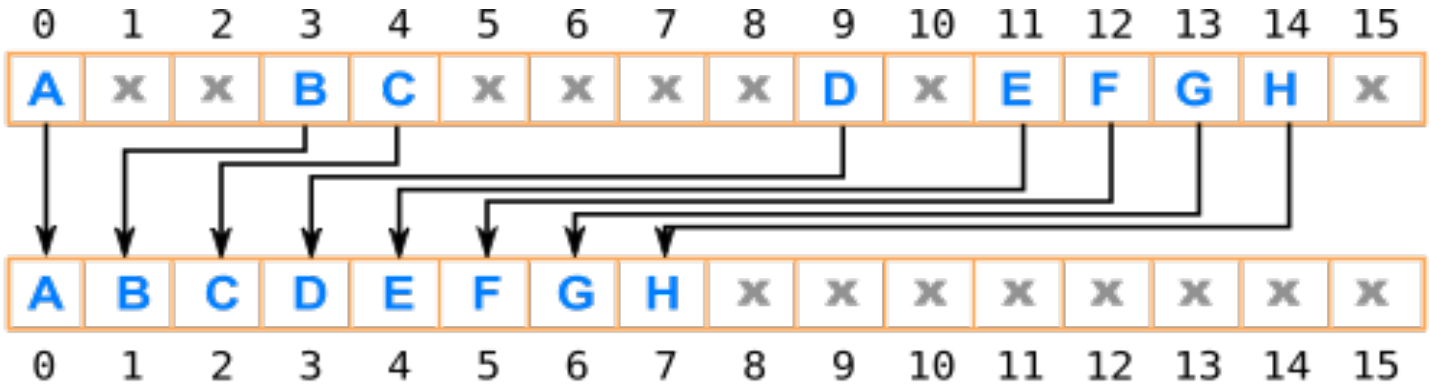
Parallelism

- Programming massively parallel systems
- Parallelizing algorithms
- Our research on 3 key components:
  1. Stream compaction
  2. Prefix Sum – 30% faster than CUDPP 1.1
  3. Sorting – faster than newest CUDPP 1.1 July 2009

3x faster than any other implementation we know of

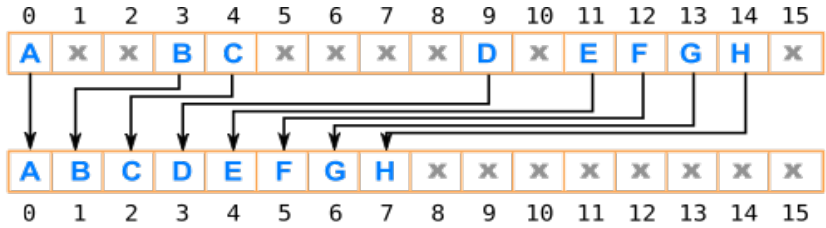
Parallelism

- Programming massively parallel systems
- Parallelizing algorithms
- Our research on 3 key components:
  1. Stream compaction
  2. Prefix Sum
  3. Sorting



Parallelism

- Programming massively parallel systems
- Parallelizing algorithms
- Our research on 3 key components:
  1. Stream compaction
  2. Prefix Sum
  3. Sorting

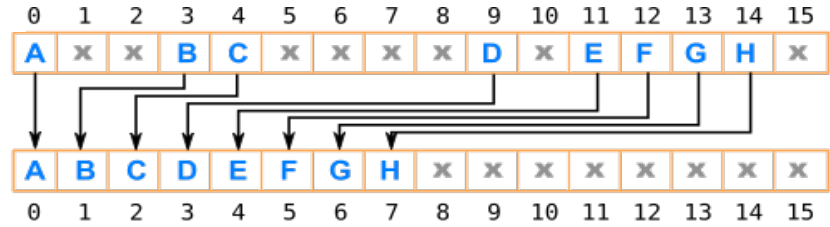


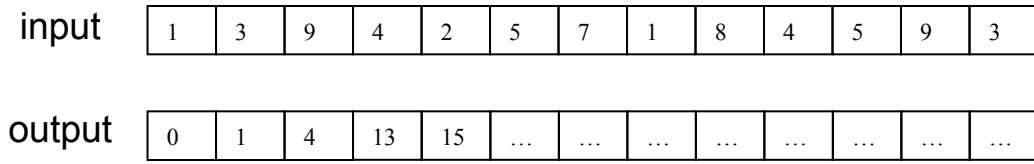
input	1	3	9	4	2	5	7	1	8	4	5	9	3
output	0	1	4	13	17	…	…	…	…	…	…	…	…

Each output element is sum of all preceding input elements

Parallelism

- Programming massively parallel systems
- Parallelizing algorithms
- Our research on 3 key components:
  1. Stream compaction
  2. Prefix Sum
  3. Sorting

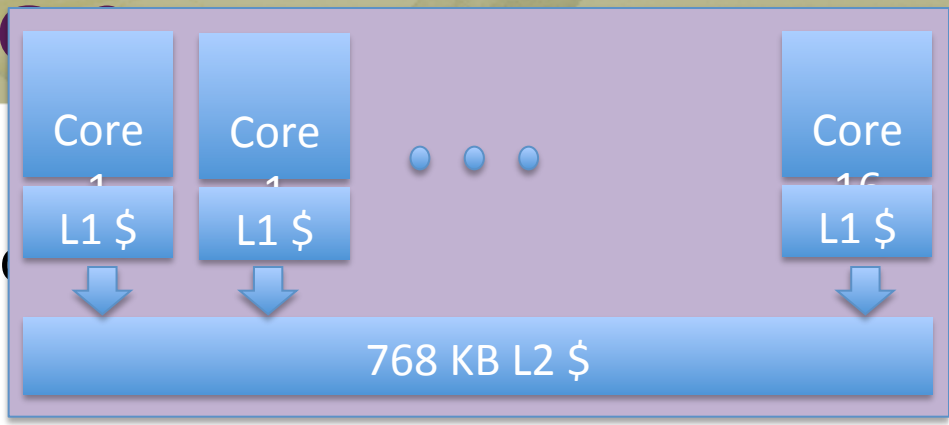




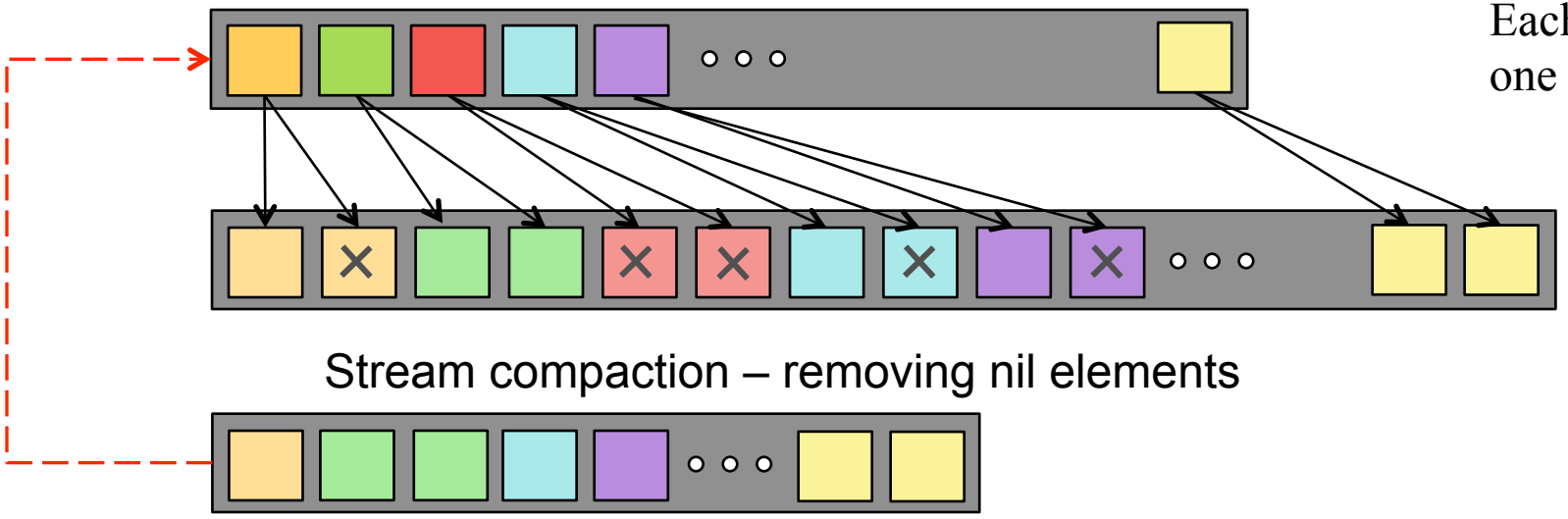
19 5 100 1 63 79

1 5 19 63 79 100

1. Stream Compacti



- Used for:
  - Load balancing & load distributi
    - Alternative to global task queue
  - Parallel Tree Traversal
    - Collision Detection - Horn, GPUGems 2, 2005.[1]



Each thread (ALU) handles one node

and outputs fixed #nodes for (possibly) continued traversal

[1] *Stream reduction operations for GPGPU applications,* Horn, GPU Gems 2, 2005.

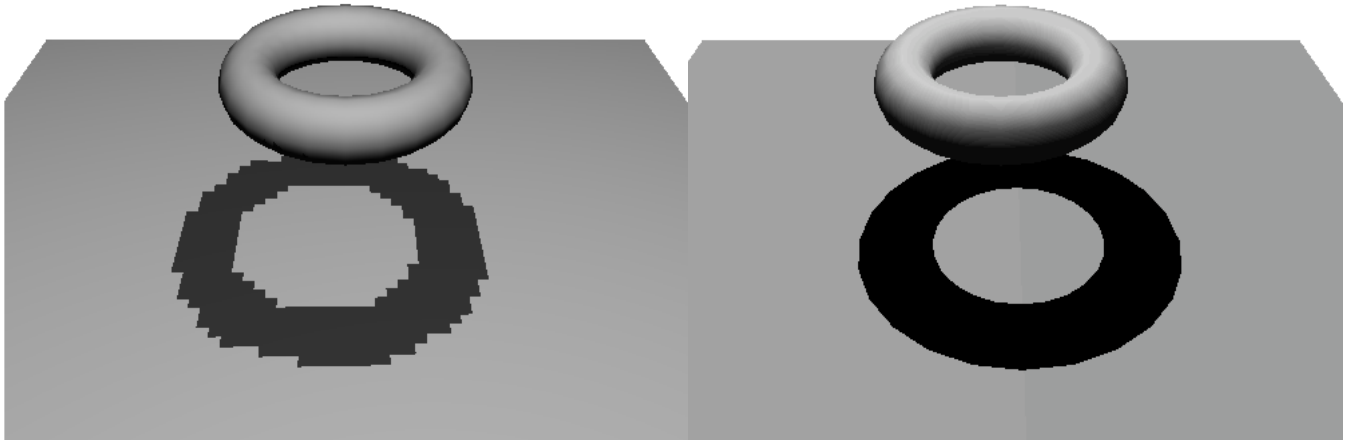
1. Stream Compaction

- Used for:
  - Load balancing & load distribution
    - Alternative to global task queue
  - Parallel Tree Traversal
    - Collision Detection - Horn, GPUGems 2, 2005.
  - Constructing spatial hierarchies
    - Lauterbach, Garland, Sengupta, Luebke, Manocha, *Fast BVH Construction on GPUs*, EGSR 2009
  - Radix Sort
    - Satish, Harris, Garland, *Designing efficient sorting algorithms for manycore GPUs*, IEEE Par. & Distr. Processing Symp., May 2009
  - Ray Tracing
    - Aila and Laine, *Understanding the Efficiency of Ray Traversal on GPUs*, HPG 2009
    - Roger, Assarsson, Holzschuch, *[2]Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU,* EGSR 2007.

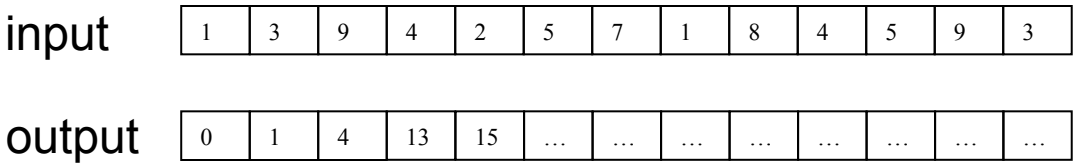
1. Stream Compaction - shadows

Alias Free Hard Shadows

- *Resolution Matched Shadow Maps*, by Aaron Lefohn, Shubhabrata Sengupta, John Owens, Siggraph 2008
  - Prefix sum, stream compaction, sorting
- *Sample Based Visibility for Soft Shadows using Alias-free Shadow Maps*, by Erik Sintorn, Elmar Eisemann, Ulf Assarsson, EGSR 2008
  - Prefix sum



2. Prefix Sum



Each output element is sum of all preceding input elements

- Good for
  - Solving recurrence equations
  - Sparse Matrix Computations
  - Tri-diagonal linear systems
  - Stream-compaction

Stream Compaction - Idea

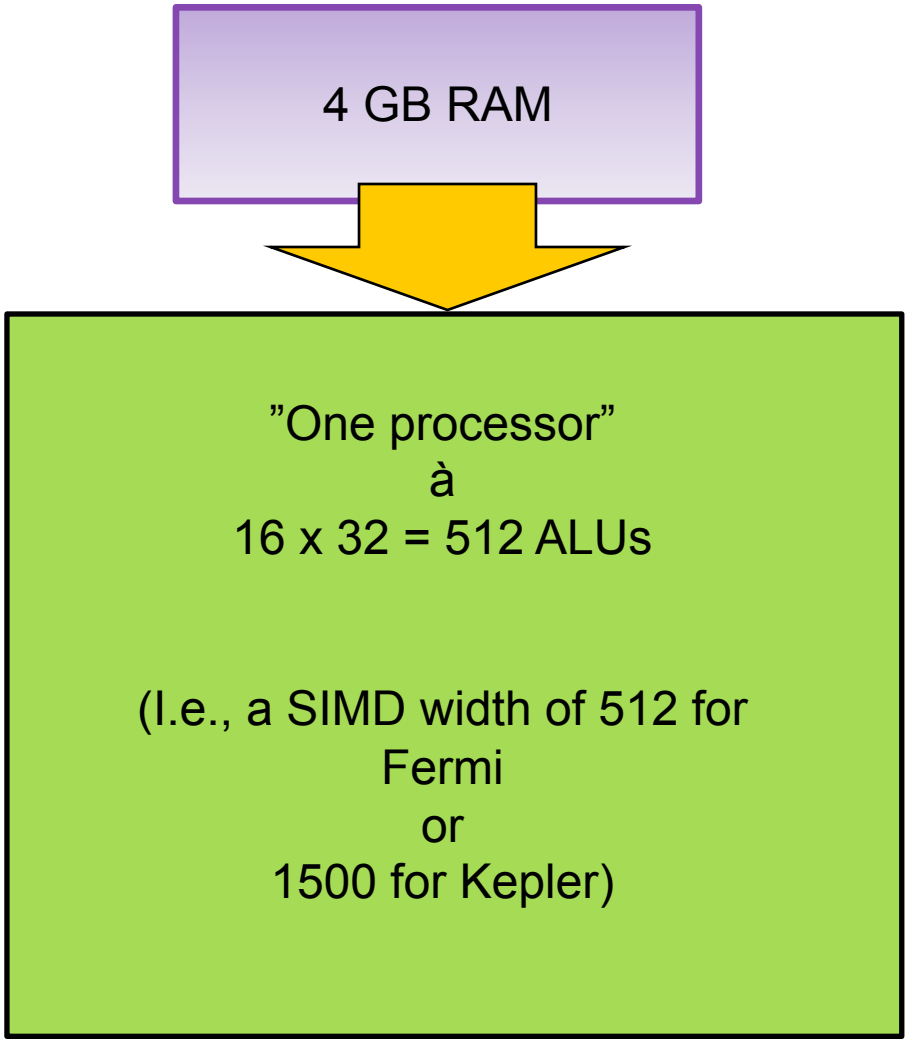
- Serial algorithms are simple
- We pay for parallelism:
  - Communication
  - Synchronization
- Don’t split problem into more parallel tasks than necessary

NVIDIA's SIMT-model

- SIMT = Single instruction multiple thread

NVIDIA calls it running one *thread* per ALU

So we have one program per 512 ALUs instead of per 32 ALUs.
=> Lowers flexibility.



We say:

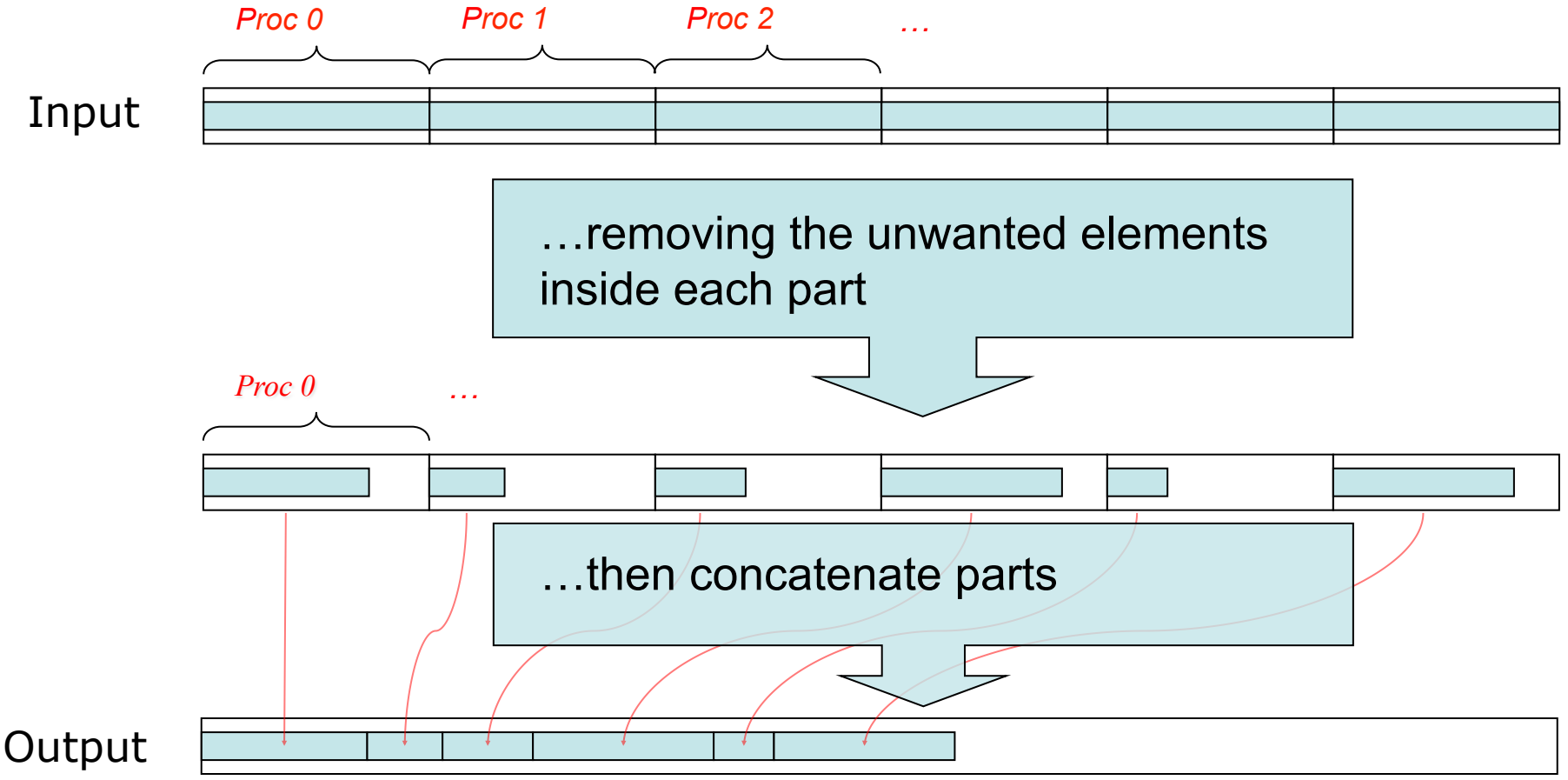
- Don't parallelize over threads!
- Parallelize over cores
  - And then over 32 threads in each core.

Stream Compaction

- Suggested by Blelloch 1990:

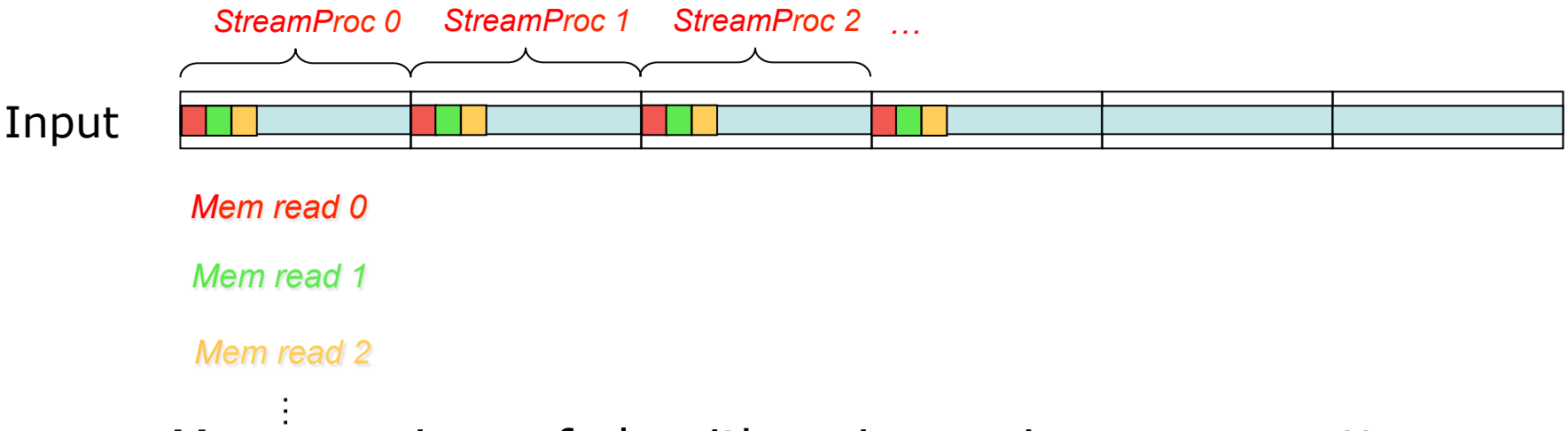
  Split input among processors and work sequentially on each part

  E.g.: Each processor sequentially compacts one part of stream



Stream Compaction

- BUT:
  - Naïvely treating each SIMD-lane a
    memory access pattern



  - Many versions of algorithms improving access pattern
  - We suggest treating hardware as a
    - Limited number of processors with a specific SIMD width
      - GTX280: 30 processors, logical SIMD width = 32 lanes
      - Fermi: 16 processors, logical SIMD width = 32 lanes

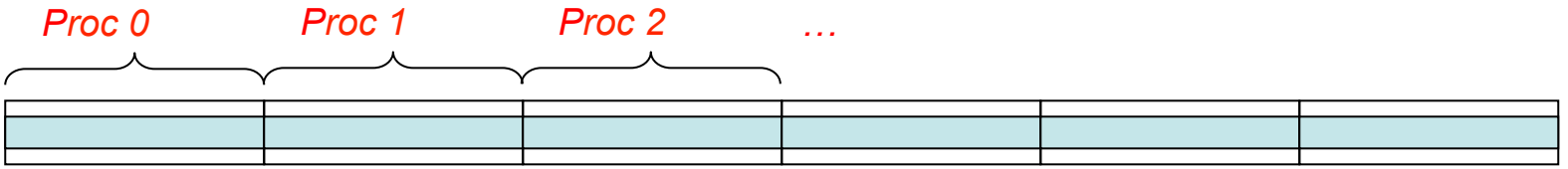
(CUDA 2.1/2.2 API)

Stream Compaction

- Our basic idea:

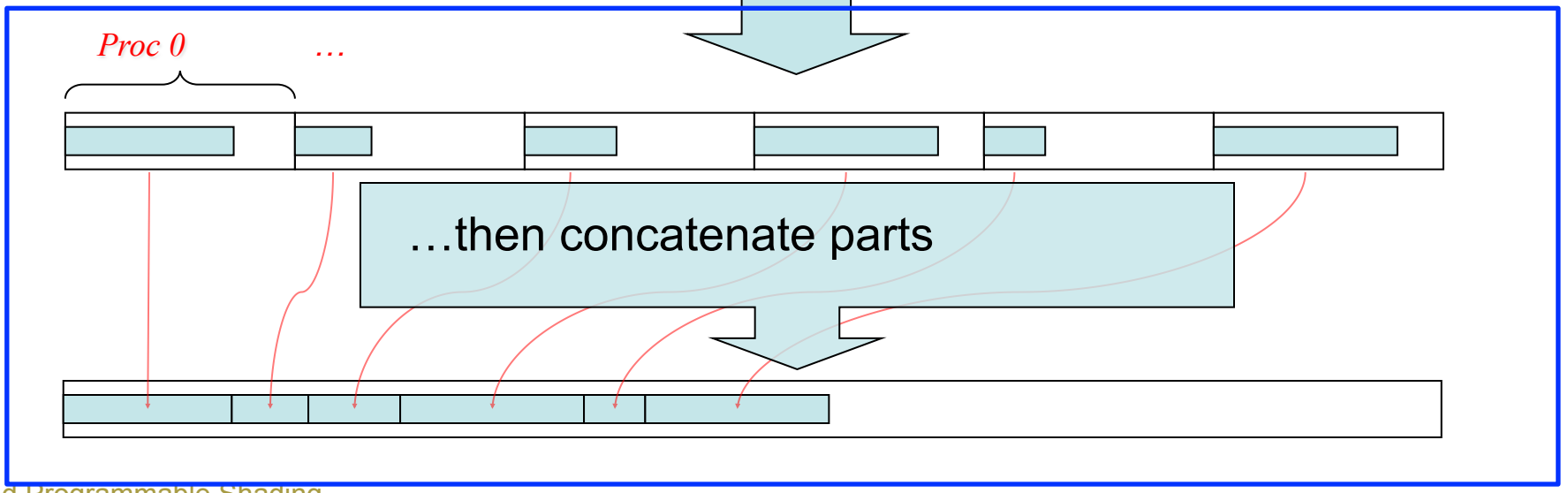
  Split input among processors and work sequentially on each part

  Each (multi-)processor sequentially compacts one part of stream

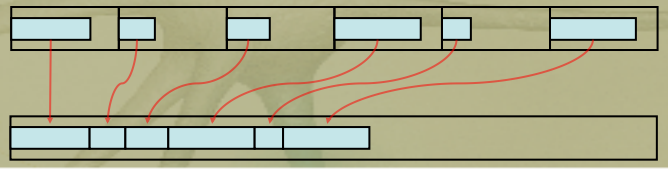


Start by computing output offsets for each processor

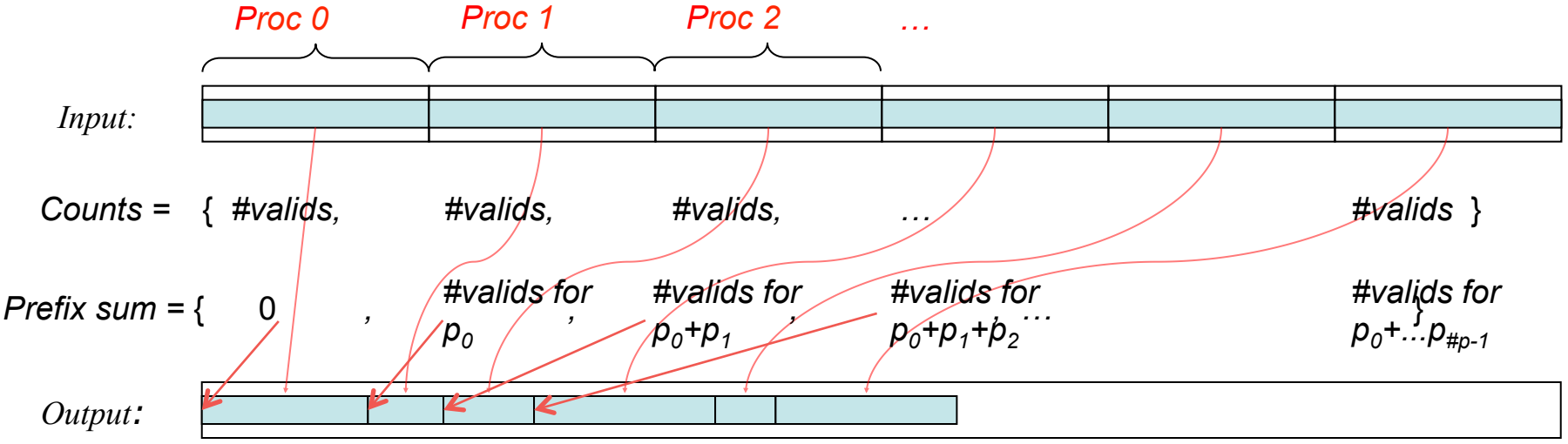
…removing the unwanted elements inside each part



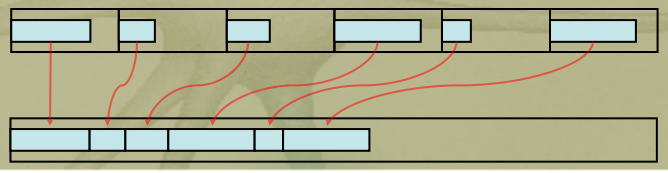
Stream Compaction



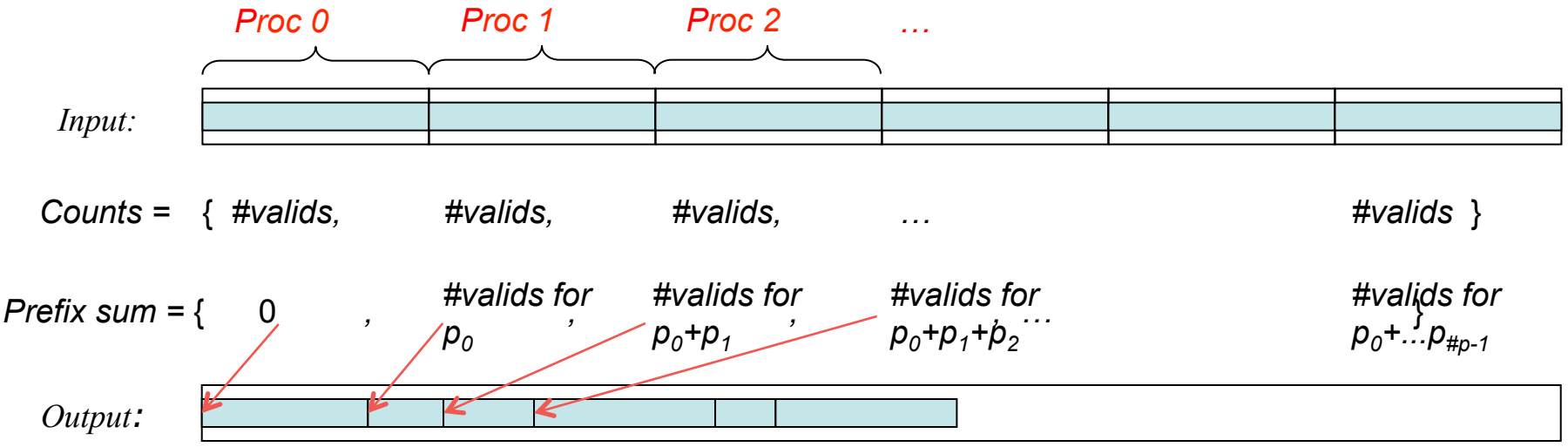
- Computing the processors' output offsets:
  - Each processor counts its number of valid elements (i.e., output length)
  - Compute Prefix Sum array for all counts
  - This array tells the output position for each processor



Stream Compaction

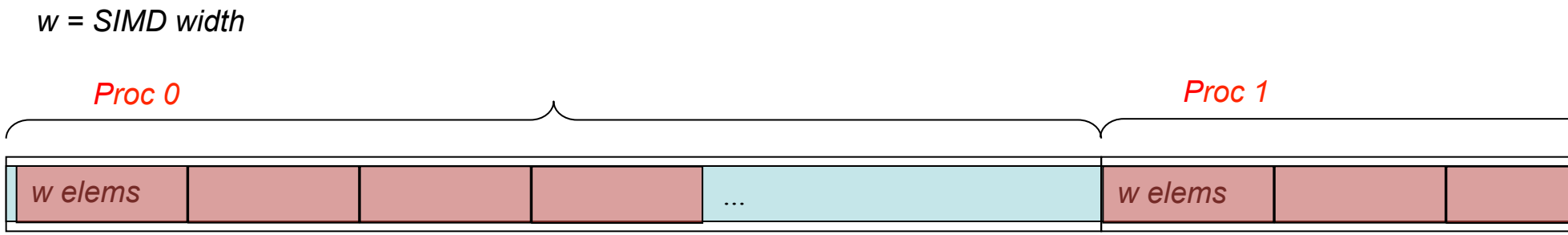


- Computing the processors' output offsets:
  - Each processor counts its number of valid elements (i.e., output length)
  - Compute Prefix Sum array for all counts
  - This array tells the output position for each processor



Stream Compaction

- Each processor counts its number of valid elements



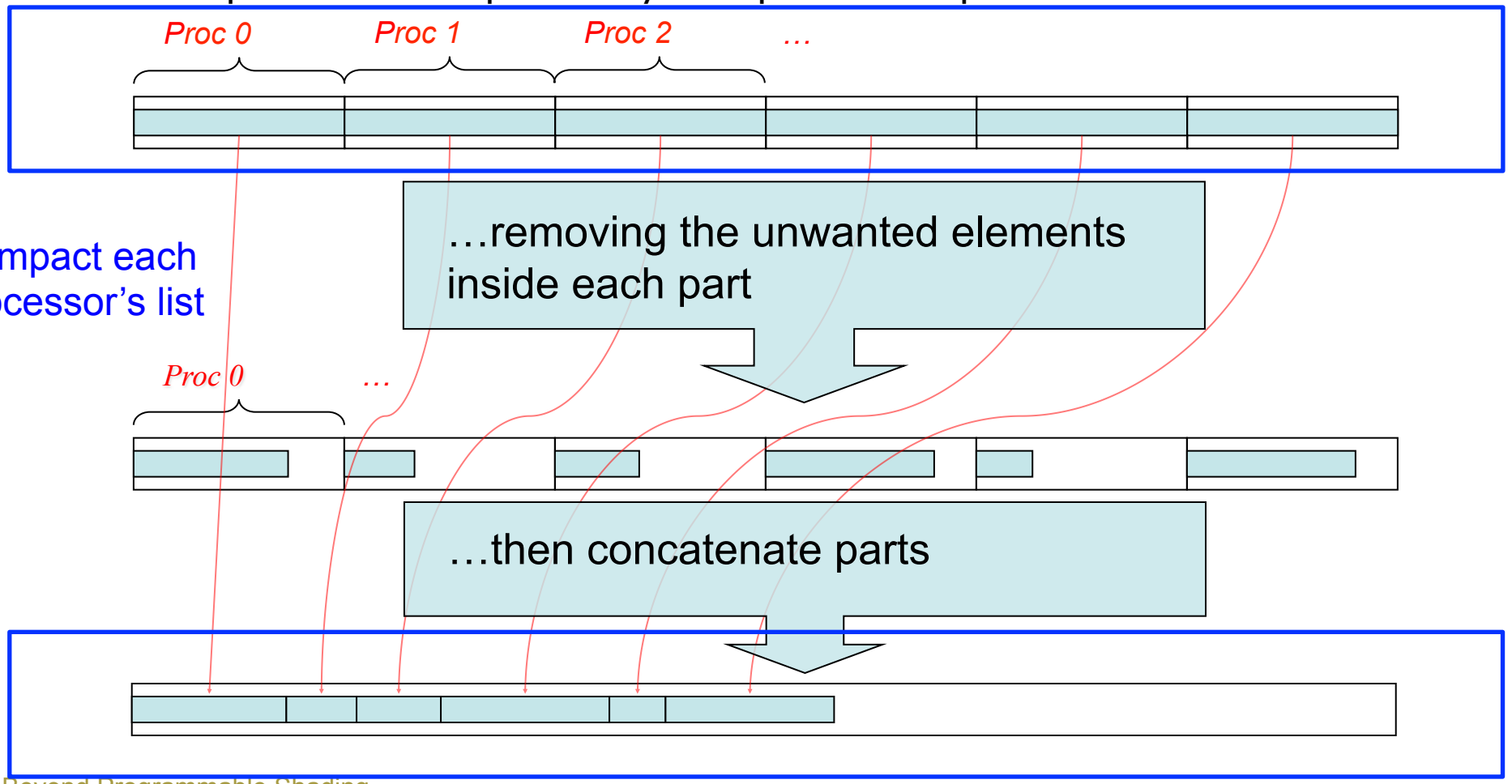
- Each processor:
  - Loop through its input list:
    - Reading *w* elements each iteration
      - *Perfectly coalesced (i.e., each thread reads 1 element)*
    - Each lane (thread / stream processor) increases its counter if its element is valid
  - *Finally, sum the w counters*

Stream Compaction

- Our basic idea:

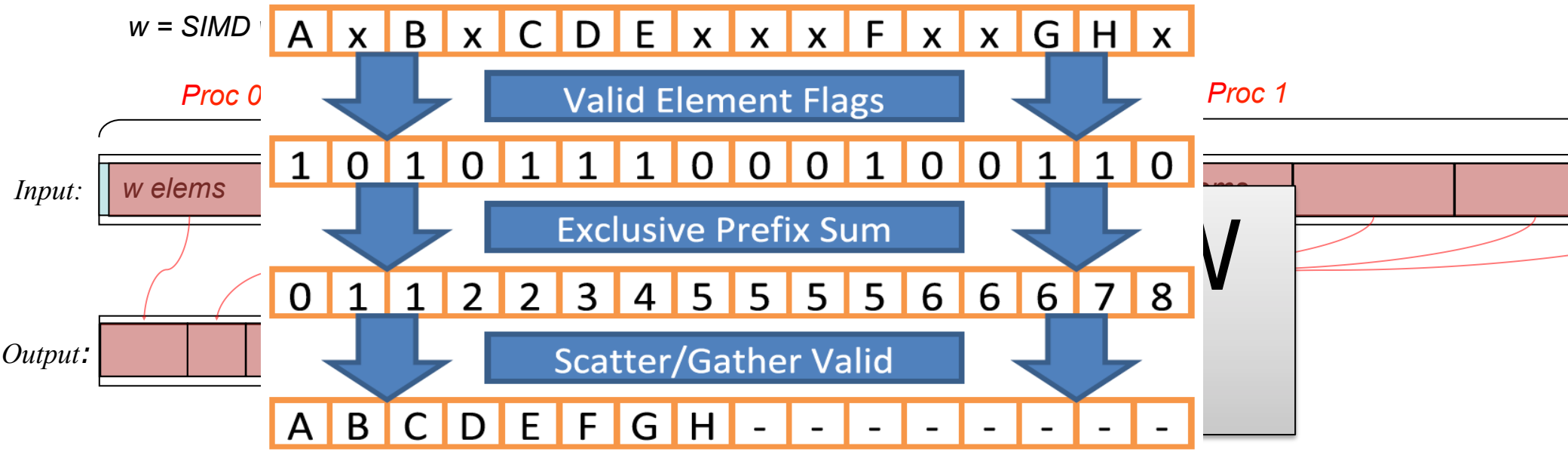
  Split input among processors and work sequentially on each part

  Each processor sequentially compacts one part of stream



Stream Compaction

- Compacting the input list for each SIMD-processor



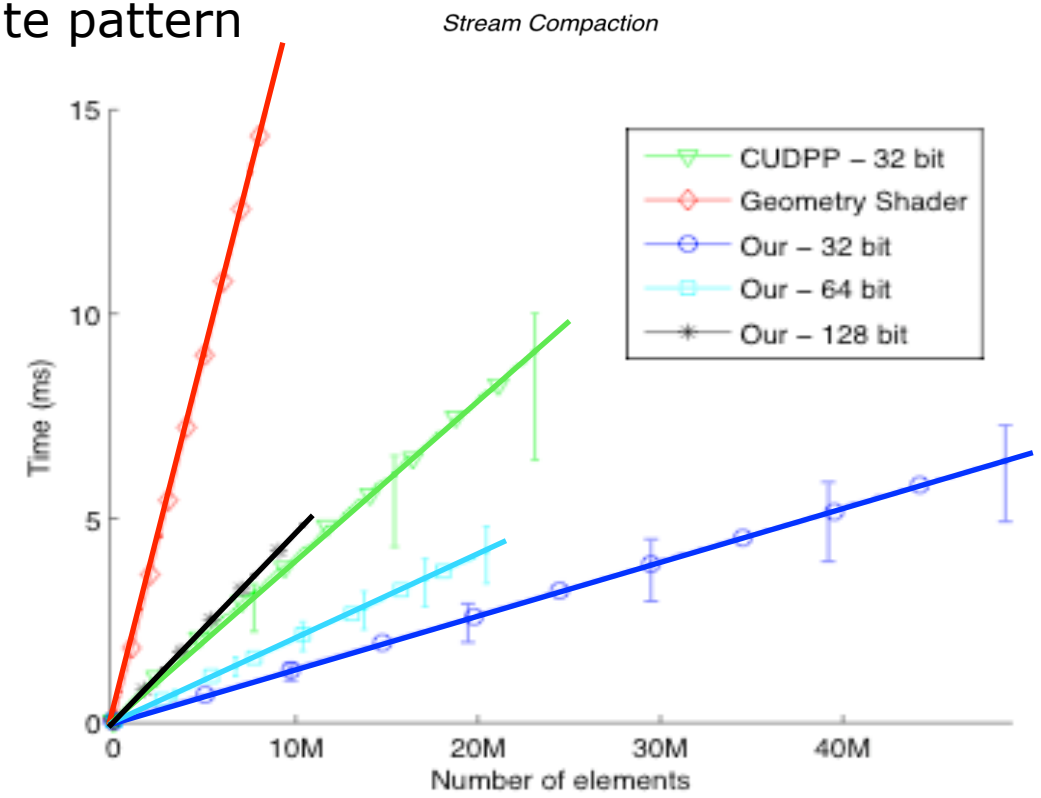
- Each processor:
  - Loop through its i
    - Reading *w* elements each iteration
      - *Perfectly coalesced (i.e., each thread reads 1 element)*
    - Use a standard parallel compaction for *w* elements
    - Write to output list and update output position by #valid elements

No bank conflicts in shared memory

Our Stream Compaction

Stream compaction with

- Optimal coalesced reads
- Good write pattern



The error bars display variations in time as the proportion of valid elements is changed. The graphs represent the average time for varying proportions of valid elements. Also shown are curves for compaction of 64 bit and 128 bit elements.

Steam Compaction

- In reality we use:
  - GTX280:
    - P = 480 to increase occupancy and hide mem latency
      - 30x4 blocks à 4 warps
        - à 32 threads
        - Hardware specific
    - Highest memory bandwidth if each lane fetches 32 bit data in 64 bit units (i.e., 2 floats instead of 1).
      - Hardware specific

32x	32 bit fetches	64 bit fetches	128 bit fetches
Bandwidth (GB/s)	77.8	102.5	73.4

Stream Compaction

- The evolution of stream compaction algorithms:

	Algorithm	4M elements	NVIDIA
shaders	*Horn (2005)*	60 ms	Geforce 8800
shaders	*..modified with Blelloch's prefix sum*	37.2 ms	Geforce 8800
shaders	*Roger, Assarsson, Holzschuch (2007)*	13.7 ms	Geforce 8800
CUDA	*Ziegler, Tevs, Theobalt, Seidel (2006)*	3.56	Geforce 8800
CUDA		2.54 ms	GTX280
CUDA	*CUDPP[1] (2009)*	1.81 ms	GTX280
CUDA	*Billeter, Olsson, Assarsson (2009)*	0.56 ms	GTX280
	What will be next… ?		

[1]CUDPP: Mark Harris, John D. Owens, Shubhabrata Sengupta, Yao Zhang, Andrew Davidson.

- *Harris, Sengupta, and Owens. "Parallel Prefix Sum (Scan) with CUDA". GPU Gems 3, 2007.*
- *Sengupta, Harris, Zhang, and Owens. "Scan Primitives for GPU Computing". Graphics Hardware 2007.*

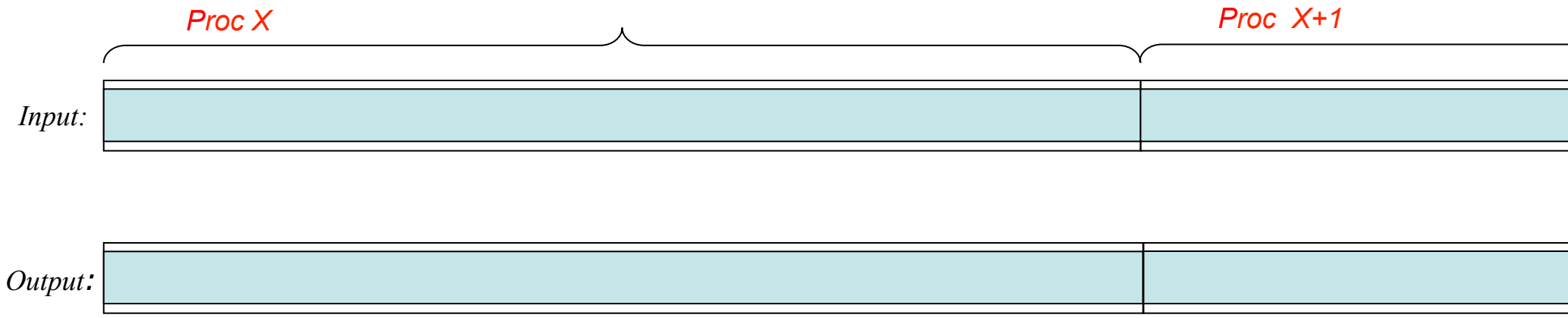
Stream Compaction

- Our Trick:
  - Split input into many independent pieces, apply sequential algorithm to each piece and combine the results later
    - Divide work among independent multi-processors
    - Use SIMD-sequential algorithm on a multi-processor

      i.e., fetch block of *w* elements

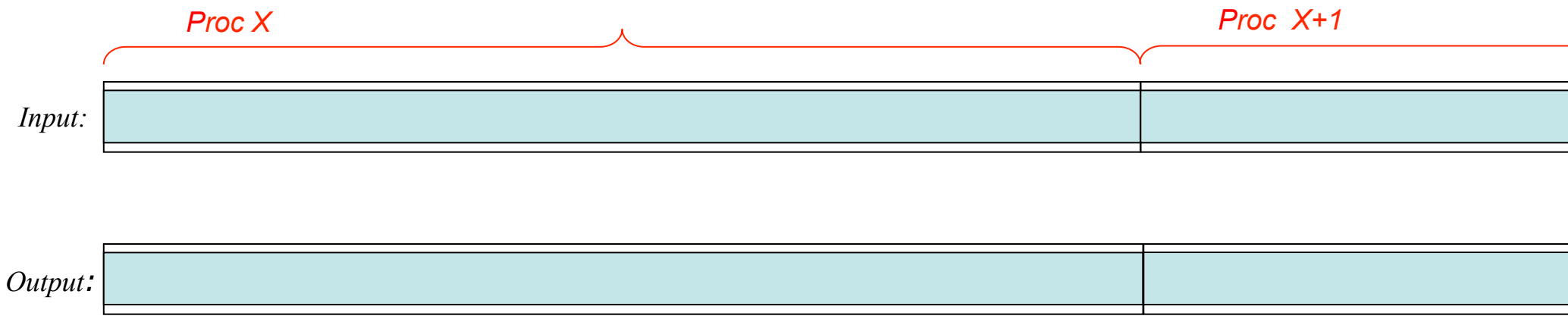
      Use parallel algorithm when working with the *w* elements

      - Work in fast shared memory



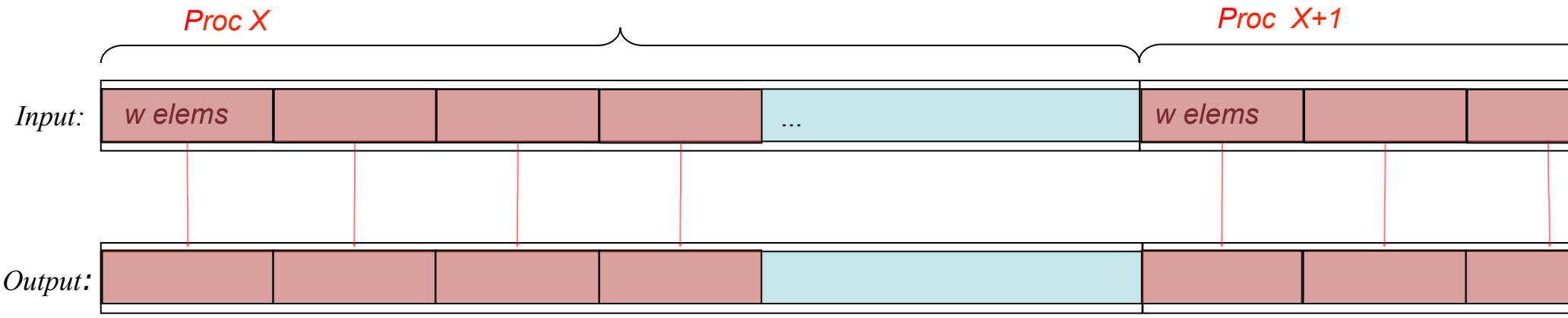
Stream Compaction

- Our Trick:
  - Split input into many independent pieces, apply sequential algorithm to each piece and combine the results later
    - Divide work among independent multi-processors
    - Use SIMD-sequential algorithm on a multi-processor
      - i.e., fetch block of *w* elements
      - Use parallel algorithm when working with the *w* elements
        - Work in fast shared memory



Stream Compaction

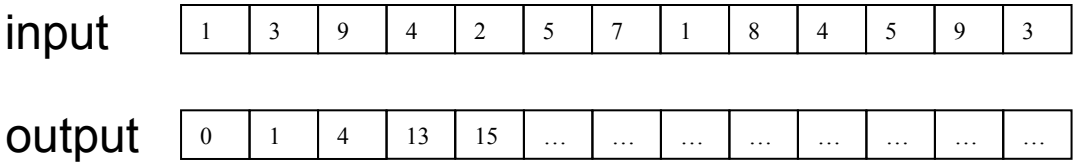
- Our Trick:
  - Split input into many independent pieces, apply sequential algorithm to each piece and combine the results later
    - Divide work among independent multi-processors
    - Use SIMD-sequential algorithm on a multi-processor
      - i.e., fetch block of *w* elements
      - Use parallel algorithm when working with the *w* elements
        - Work in fast shared memory



2. Prefix Sum

Can this philosophy be used to design a fast prefix sum

?



Prefix sum: Each output element is sum of all preceding input elements

2. Prefix Sum

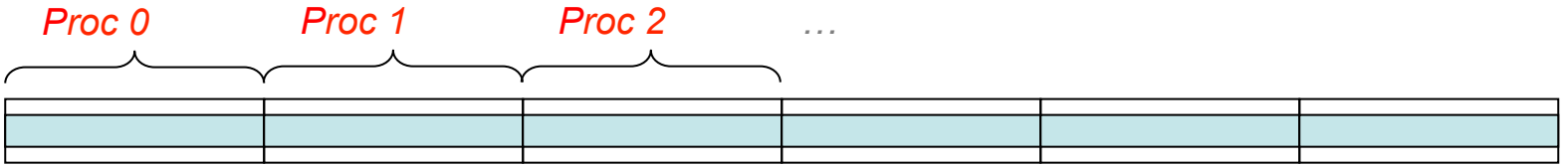
Yes!

Explained in the slides, available at:
http://www.cse.chalmers.se/~uffe/publications.htm

Making a fast Prefix Sum

- Simple modification:

  Split input among processors

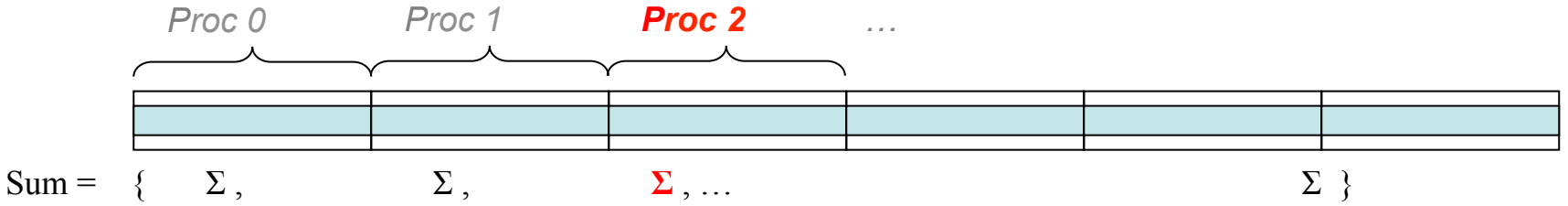


1. Each processor computes the **sum** of all its elements
2. Compute a **prefix sum** over the *p* sums
3. Each proc executes a *SIMD-sequential* **prefix sum** for its elements
   - and simultaneously adds the sum of the elements in all previous sublists

Making a fast Prefix Sum

- Simple modification:

  Split input among processors

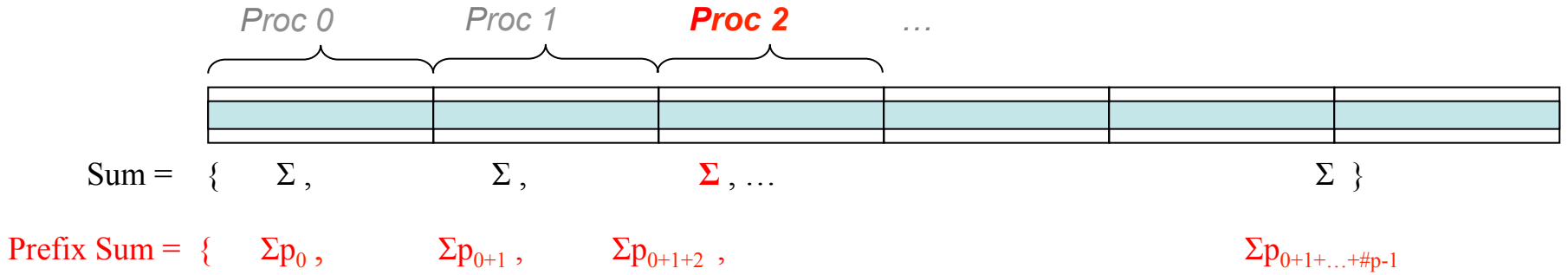


1. Each processor computes the **sum** of all its elements
2. Compute a **prefix sum** over the *p* sums
3. Each proc executes a *SIMD-sequential* **prefix sum** for its elements
   - and simultaneously adds the sum of the elements in all previous sublists

Making a fast Prefix Sum

- Simple modification:

  Split input among processors

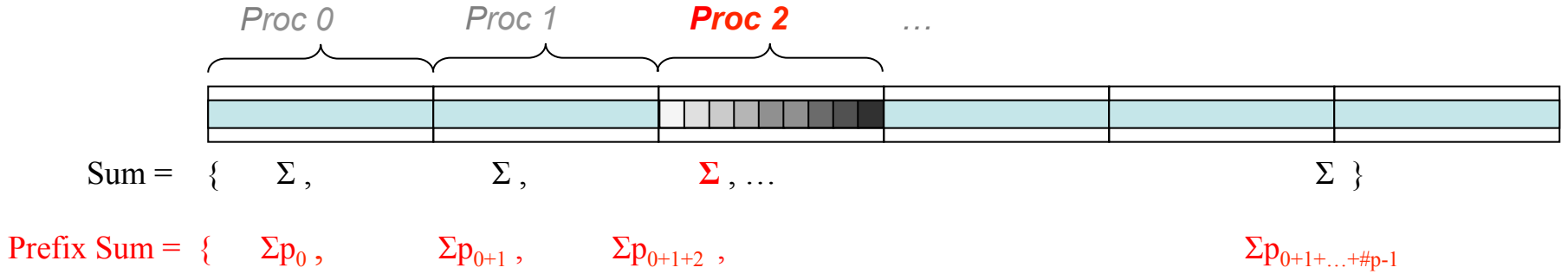


1. Each processor computes the **sum** of all its elements
2. Compute a **prefix sum** over the *p* sums
3. Each proc executes a *SIMD-sequential* **prefix sum** for its elements
   - and simultaneously adds the sum of the elements in all previous sublists

Making a fast Prefix Sum

- Simple modification:

  Split input among processors

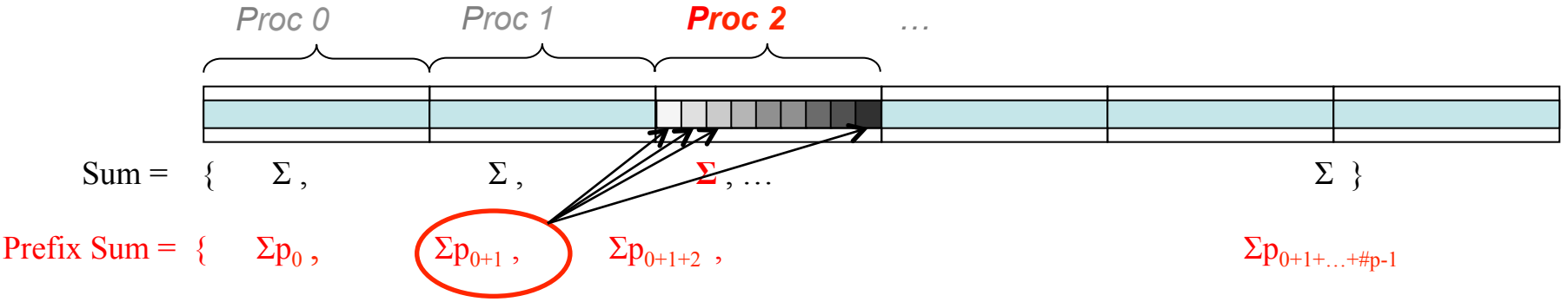


1. Each processor computes the **sum** of all its elements
2. Compute a **prefix sum** over the *p* sums
3. Each proc executes a *SIMD-sequential* **prefix sum** for its elements
   - and simultaneously adds the sum of the elements in all previous sublists

Making a fast Prefix Sum

- Simple modification:

  Split input among processors

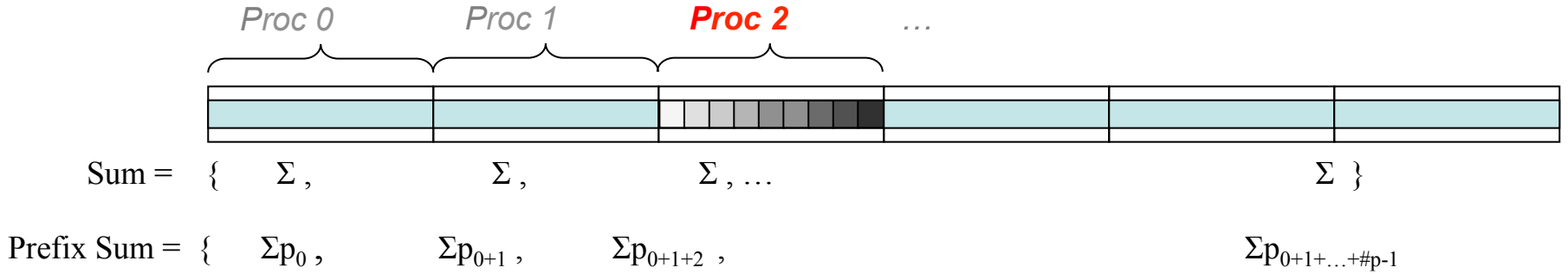


1. Each processor computes the **sum** of all its elements
2. Compute a **prefix sum** over the *p* sums
3. Each proc executes a *SIMD-sequential* **prefix sum** for its elements
   - and simultaneously **adds the sum of the elements** in all previous sublists

Making a fast Prefix Sum

- Simple modification:

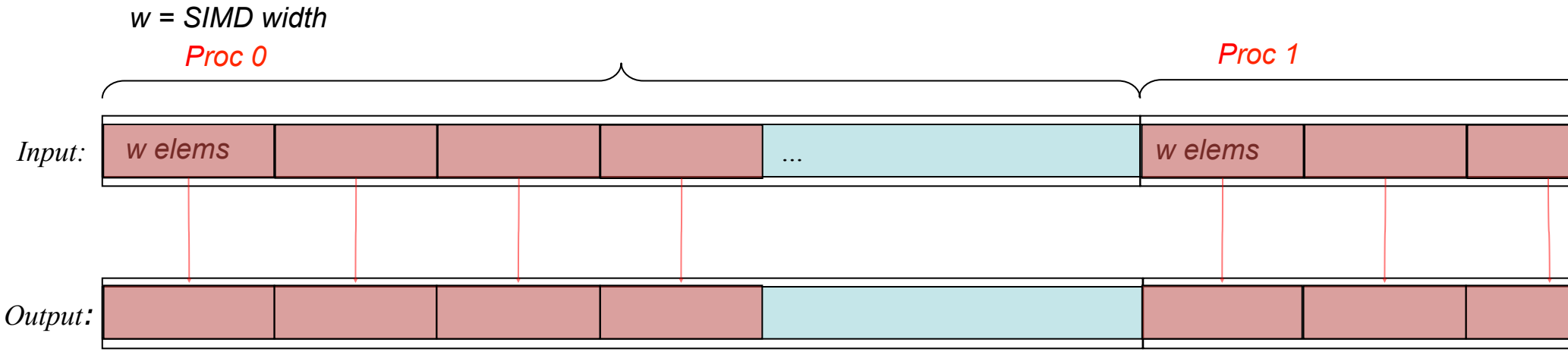
  Split input among processors



1. Each processor computes the **sum** of all its elements
2. Compute a **prefix sum** over the *p* sums
3. Each proc executes a *SIMD-sequential* **prefix sum** for its elements
   - and simultaneously adds the sum of the elements in all previous sublists

Making a fast Prefix Sum

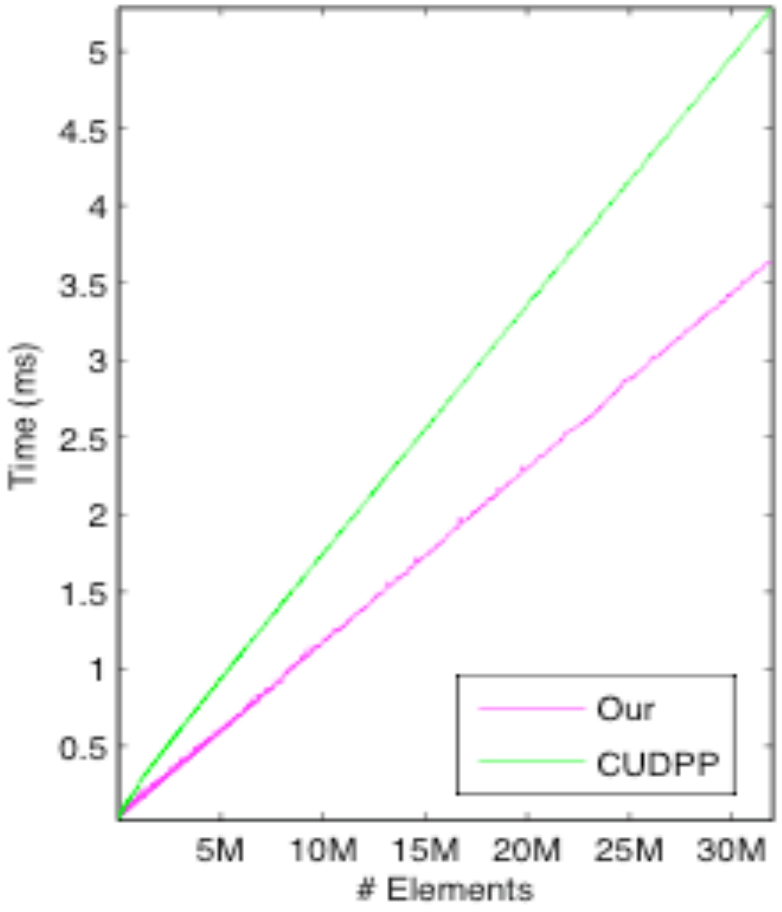
3. Each processor executes a SIMD-sequential prefix sum of all its elements:



- Each processor:
  - Loop through its input list:
    - Reading *w* elements each iteration
      - *Perfectly coalesced (i.e., each thread reads 1 element)*
    - Compute a standard parallel prefix sum for *w* elements
      - and simultaneously adds the sum of prev elements incl all previous sublists
    - Write to output list
      - Perfectly coalesced
    - Update this sum by last element of *w*

Results: Prefix Sum

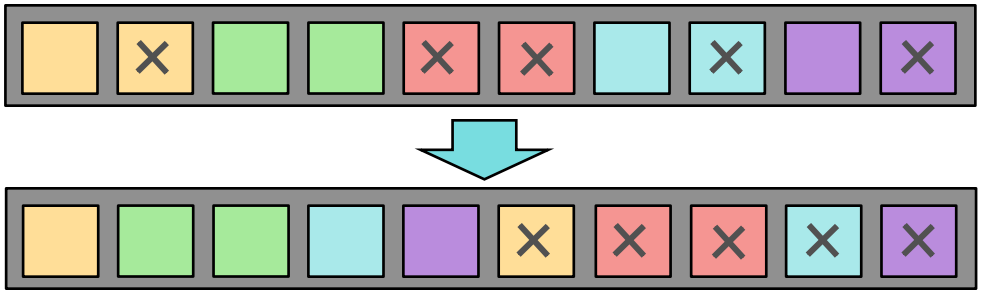
- Easier than compaction
  - Number of output elements is equal to inputs
  - ⇒ perfect coalescing when reading and writing!
- Results: 32bit elements
  - 50 microseconds/M elements



32M elements:

	GPU	Time
Our	GTX280	3.7 ms
CUDPP	GTX280	5.3 ms

Radix Sort



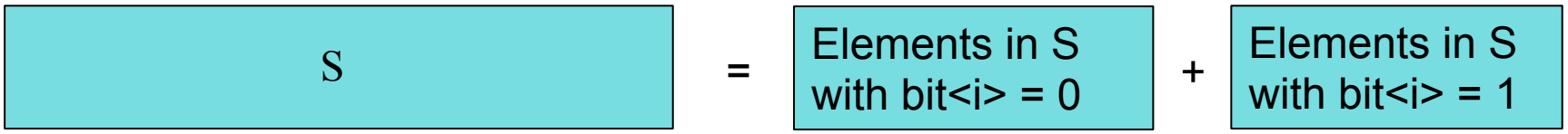
- "Stream split"
  - Like compaction
  - Place invalid elements at the end of the output buffer
    - (After all the valid elements)
- Radix Sort
  - Apply stream split once for each bit in the key

Radix Sort

- Radix sorting a stream of n 32-bits elements:

S = Stream of n elements

For i=0..31



Using stream split

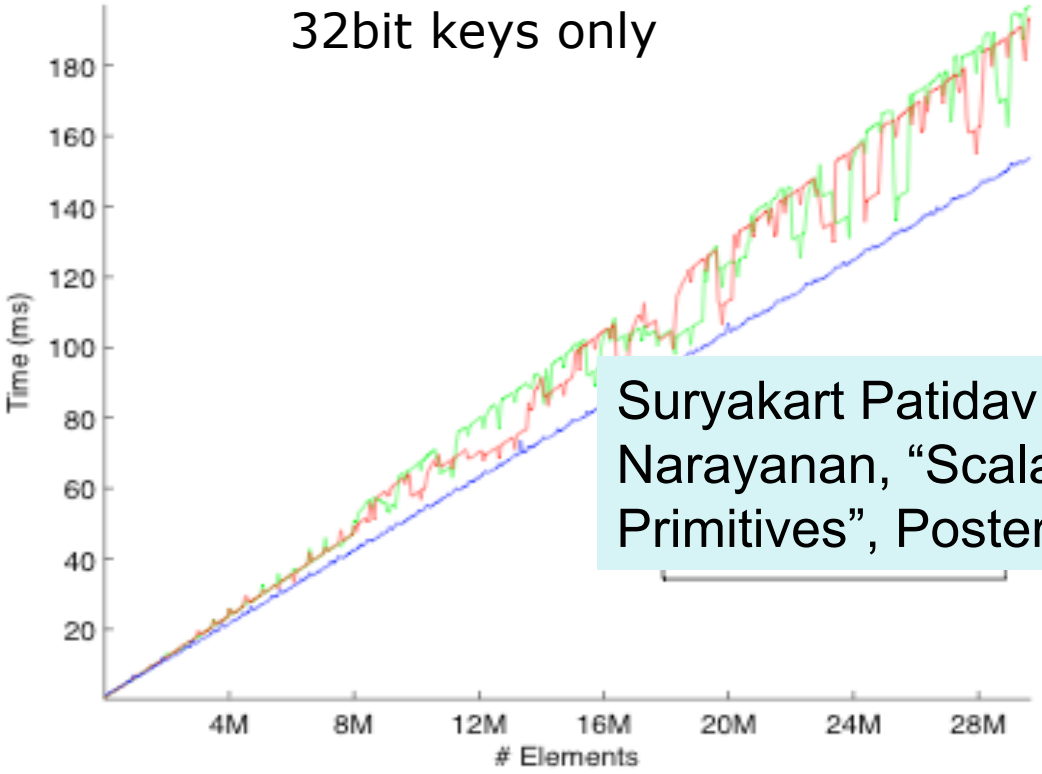
Only 32 invocations of the stream split function

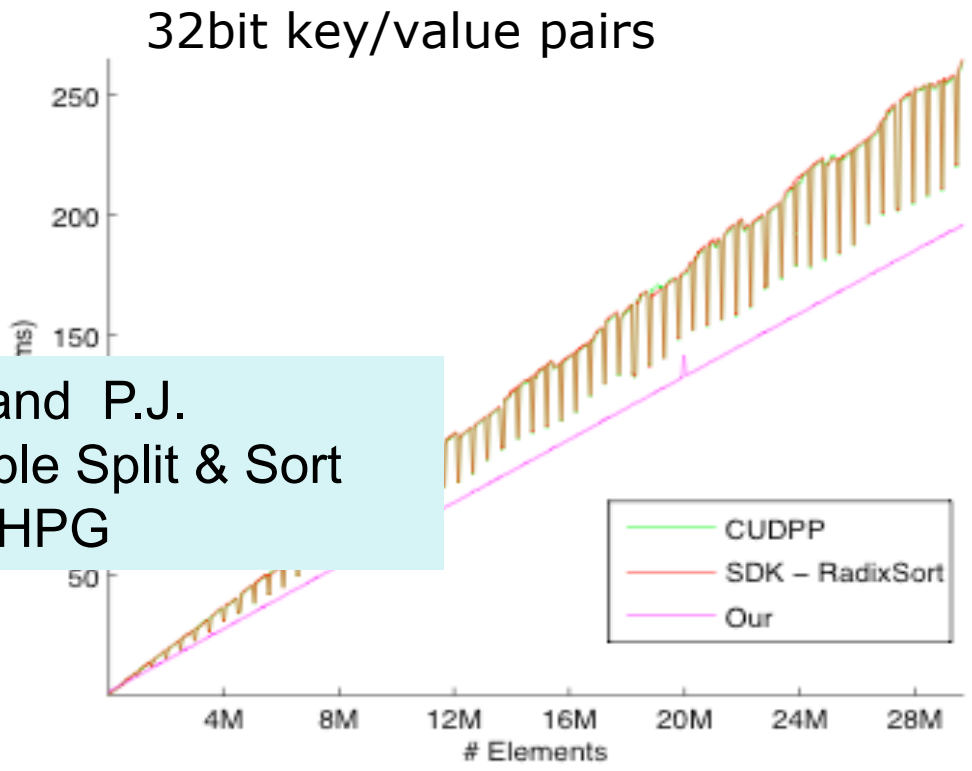
- Internal order of valid/invalid elements must be preserved in each split

Result: sorting 4M 32-bits elements (key/value) in 29 ms, GTX280.

Today: sorting ~1M elements per millisecond (keys)

Radix Sort



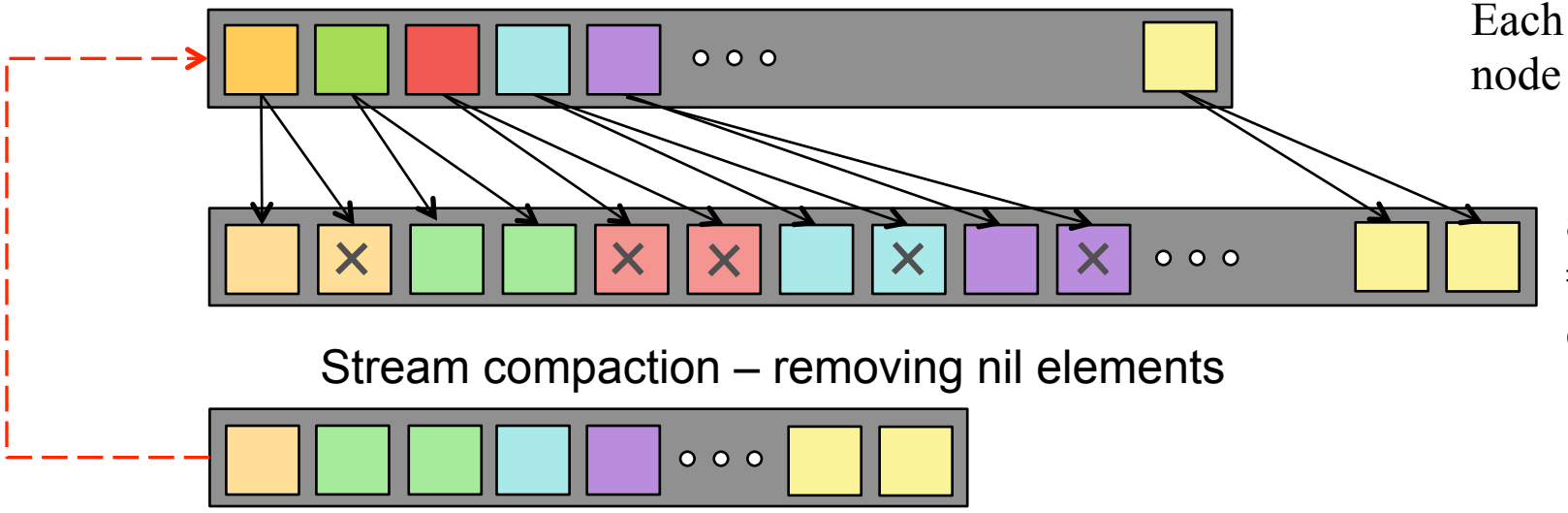


Suryakart Patidav and P.J. Narayanan, “Scalable Split & Sort Primitives”, Poster HPG

Code downloadable here: www.cse.chalmers.se/~billeter/pub/pp

Parallel Tree Traversal

Breadth first faster than depth first



Each processor handles one node

and outputs fixed #nodes for (possibly) continued traversal

[1] *Stream reduction operations for GPGPU applications,* Horn, GPU Gems 2, 2005.