Lecture 8: GPU Programming

CSE599G1: Spring 2017

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

Announcements

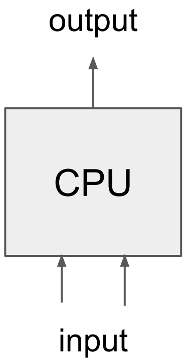
- **Project proposal** due on Thursday (4/28) 5pm.

- **Assignment 2** will be out today, due in two weeks.
  - Implement GPU kernels and use cublas library
  - Infer output shapes and memory planning

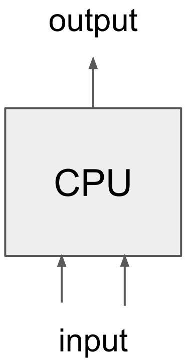
Overview

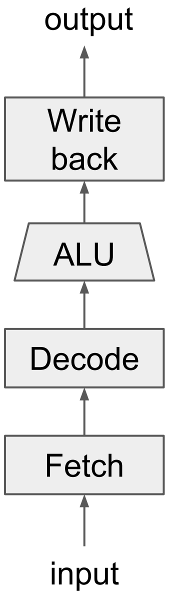
- GPU architecture
- CUDA programming model
- Case study of efficient GPU kernels

CPU vs GPU

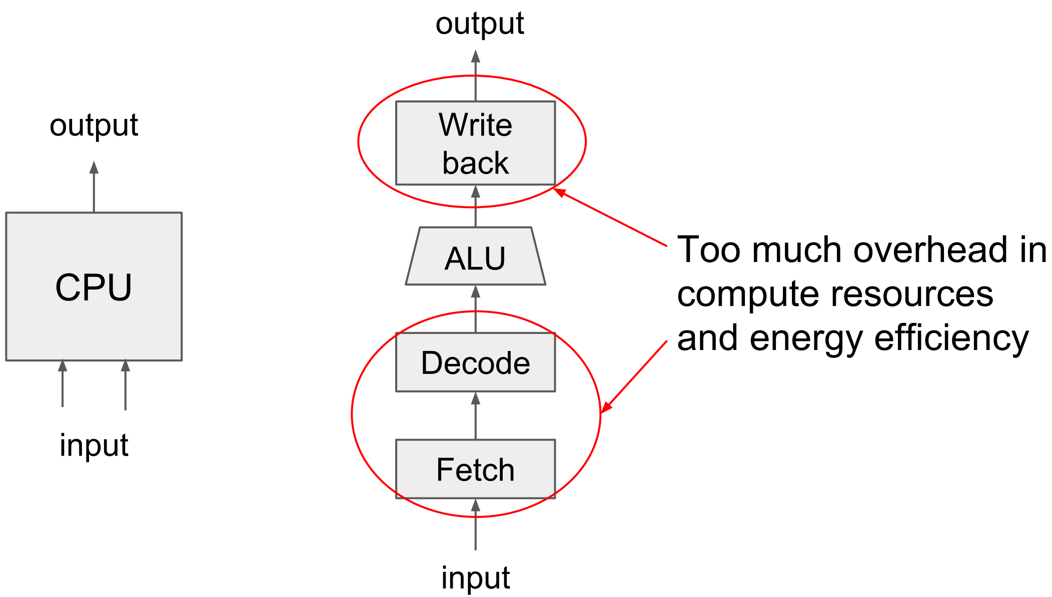


CPU vs GPU

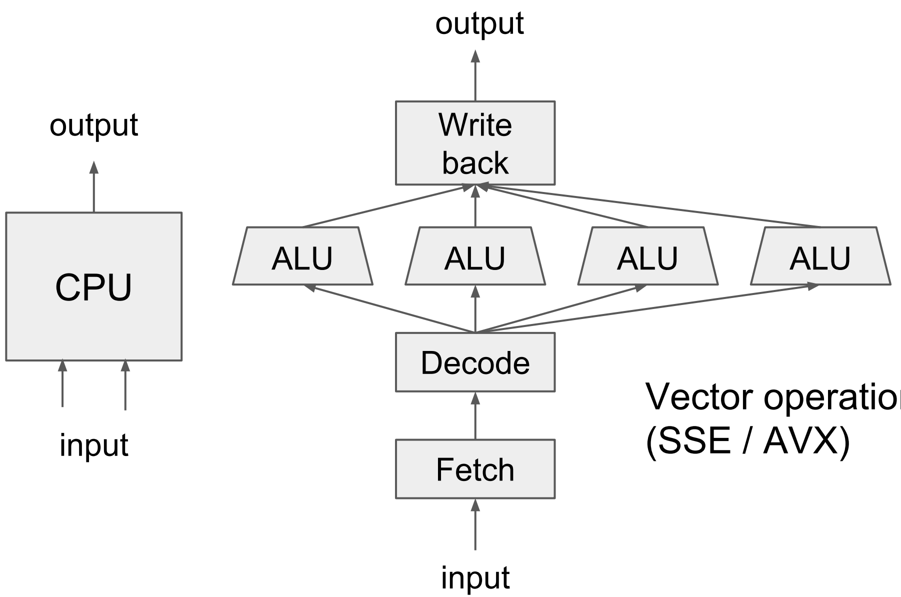




CPU vs GPU

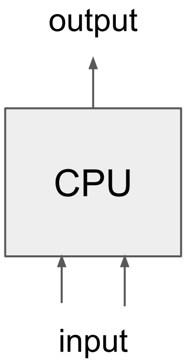

output
CPU
input
output
Write back
ALU
Decode
Fetch
input
Too much overhead in compute resources and energy efficiency

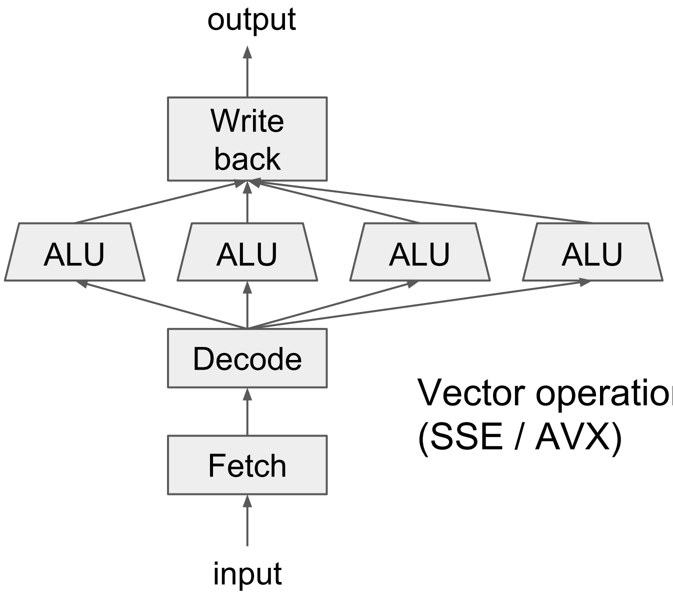
CPU vs GPU



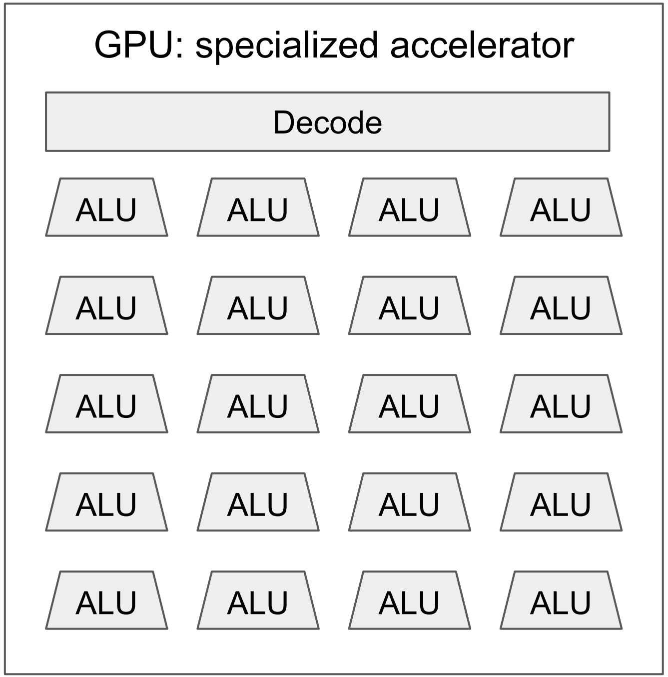
Vector operations (SSE / AVX)

CPU vs GPU





Vector operations
(SSE / AVX)



Streaming Multiprocessor (SM)

Decoder: pick the next instructions

Registers

SP float core

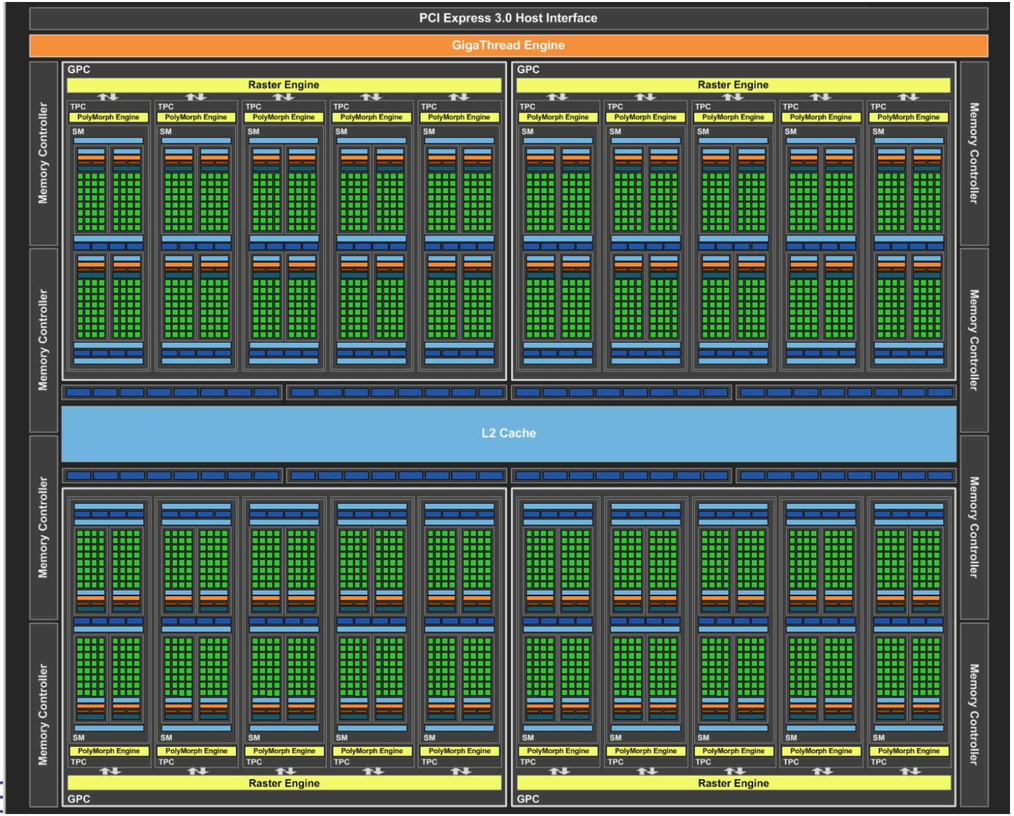
DP float core

Load/store memory

Special function unit

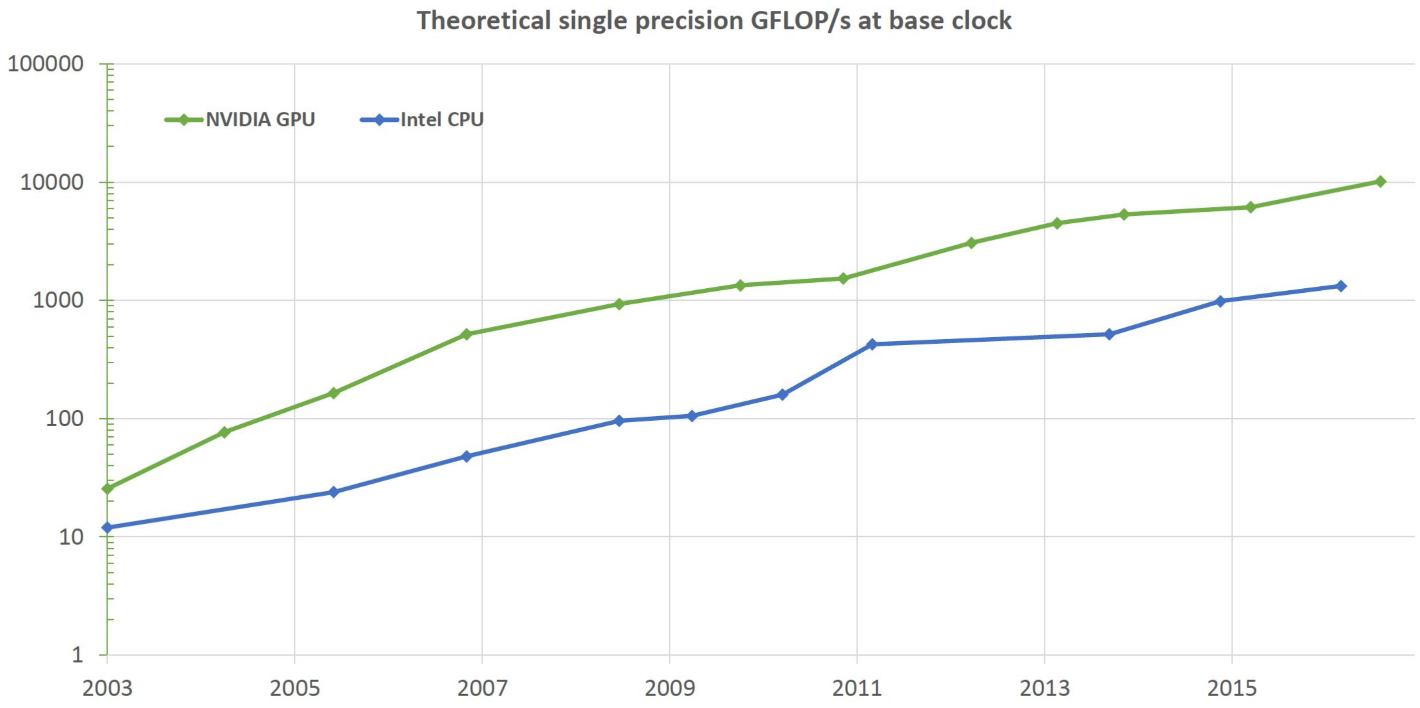
Multiple caches

GPU Architecture



PAUL G. ALLE
OF COMPUTER SCIENCE & ENGINEERING

Theoretical peak FLOPS comparison

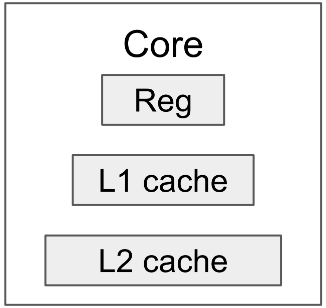


PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

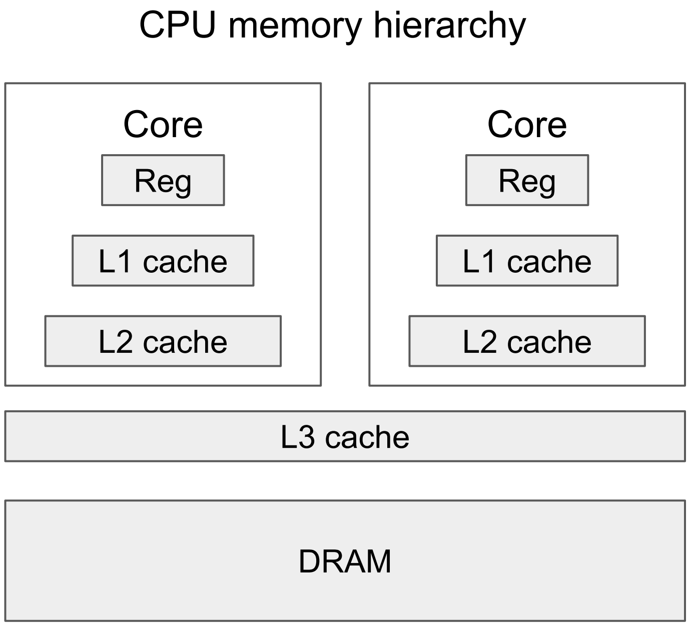
* https://github.com/oxford-cs-deepnlp-2017/lectures/blob/master/Lecture%206%20-%20Nvidia%20RNNs%20and%20GPUs.pdf

Memory Hierarchy

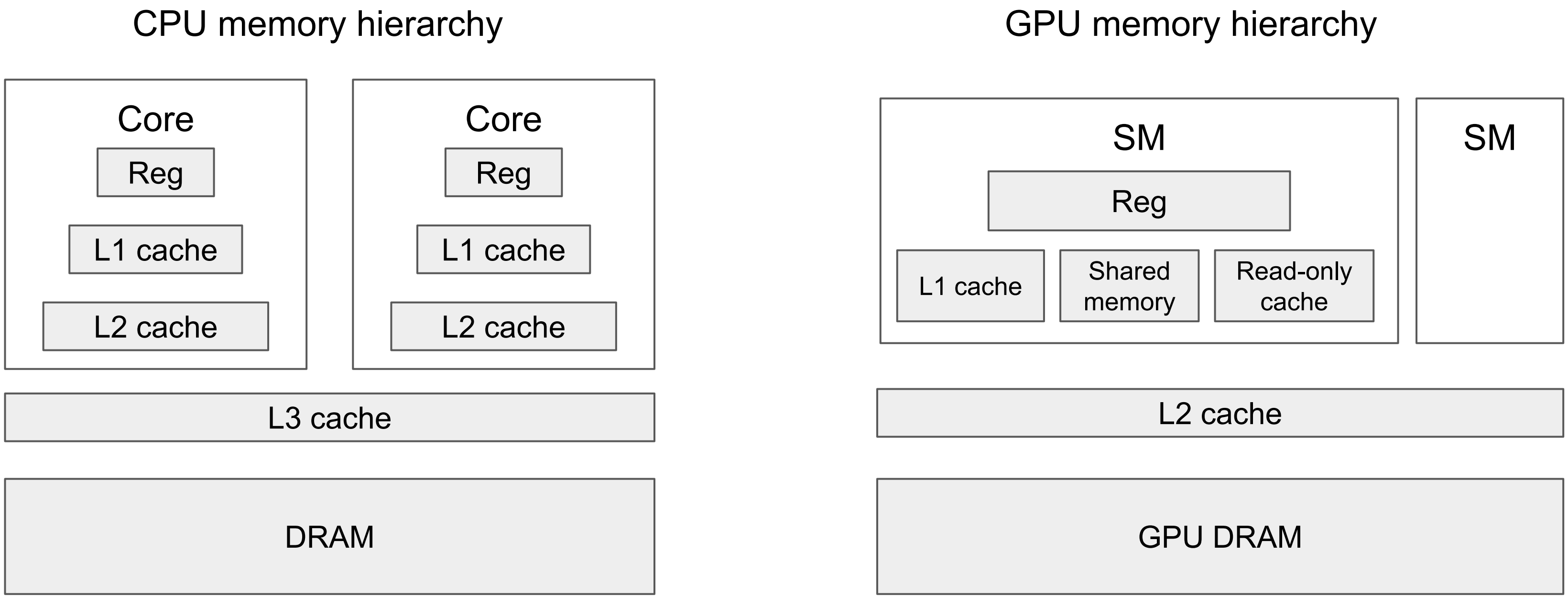
CPU memory hierarchy



Memory Hierarchy

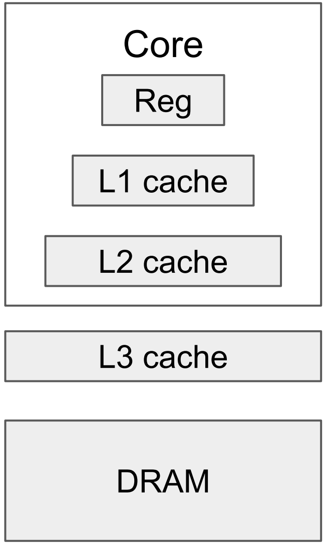


Memory Hierarchy



Memory Hierarchy

CPU memory hierarchy



Intel Xeon E7-8870v4
Cores: 20
Reg / core: ??

L1 / core: 32KB

L2 / core: 256KB

L3 cache: 50MB

DRAM: 100s GB

Price: $12,000

GPU memory hierarchy

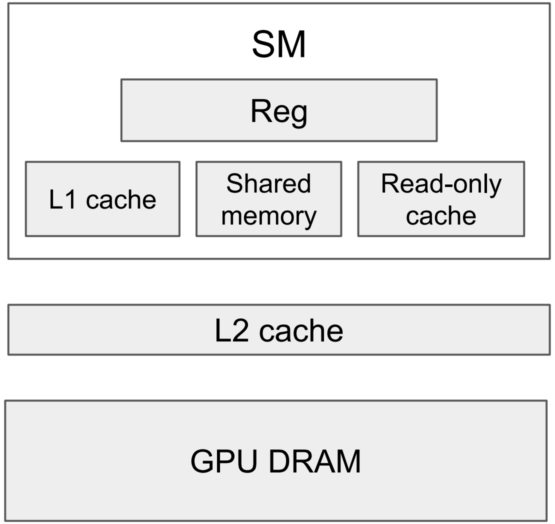
Titan X Pascal
SMs: 28
Cores / SM: 128
Reg / SM: 256 KB

L1 / SM: 48 KB
Sharedmem / SM: 64 KB

L2 cache: 3 MB

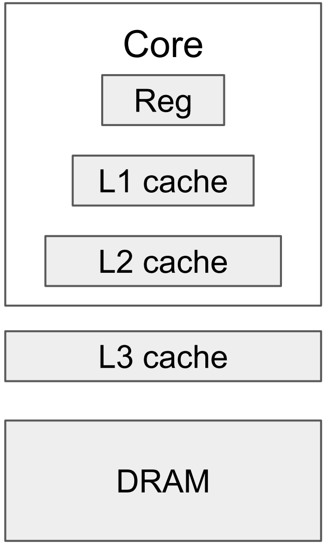
GPU DRAM: 12 GB

Price: $1,200



Memory Hierarchy

CPU memory hierarchy



Intel Xeon E7-8870v4
Cores: 20
Reg / core: ??

L1 / core: 32KB

L2 / core: 256KB

L3 cache: 50MB

DRAM: 100s GB

Price: $12,000

GPU memory hierarchy

Titan X Pascal
SMs: 28
Cores / SM: 128
Reg / SM: 256 KB

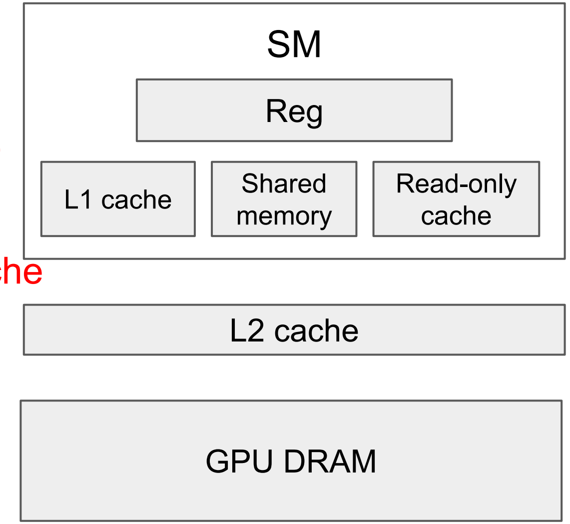
L1 / SM: 48 KB
Sharedmem / SM: 64 KB

More registers than L1 cache

L2 cache: 3 MB

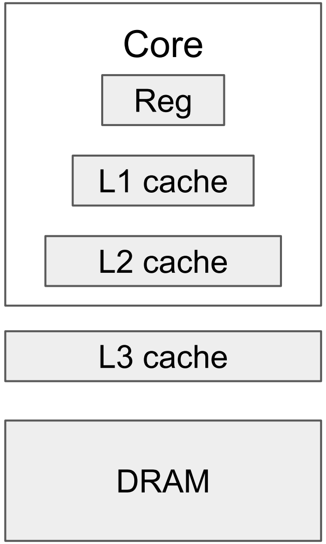
GPU DRAM: 12 GB

Price: $1,200



Memory Hierarchy

CPU memory hierarchy



Intel Xeon E7-8870v4
Cores: 20
Reg / core: ??

L1 / core: 32KB

L2 / core: 256KB

L3 cache: 50MB

DRAM: 100s GB

Price: $12,000

GPU memory hierarchy

Titan X Pascal
SMs: 28
Cores / SM: 128
Reg / SM: 256 KB

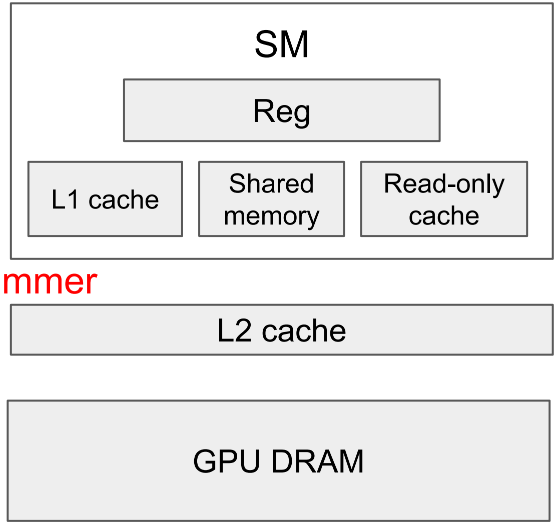
L1 / SM: 48 KB
Sharedmem / SM: 64 KB

L1 cache controlled by programmer

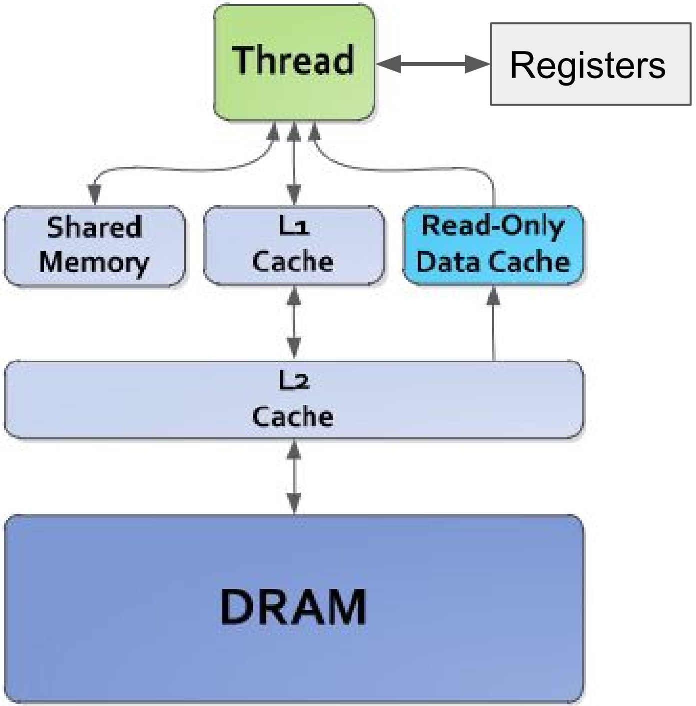
L2 cache: 3 MB

GPU DRAM: 12 GB

Price: $1,200



GPU Memory Latency



Registers: R 0 cycle / R-after-W ~20 cycles

L1/texture cache: 92 cycles
Shared memory: 28 cycles
Constant L1 cache: 28 cycles

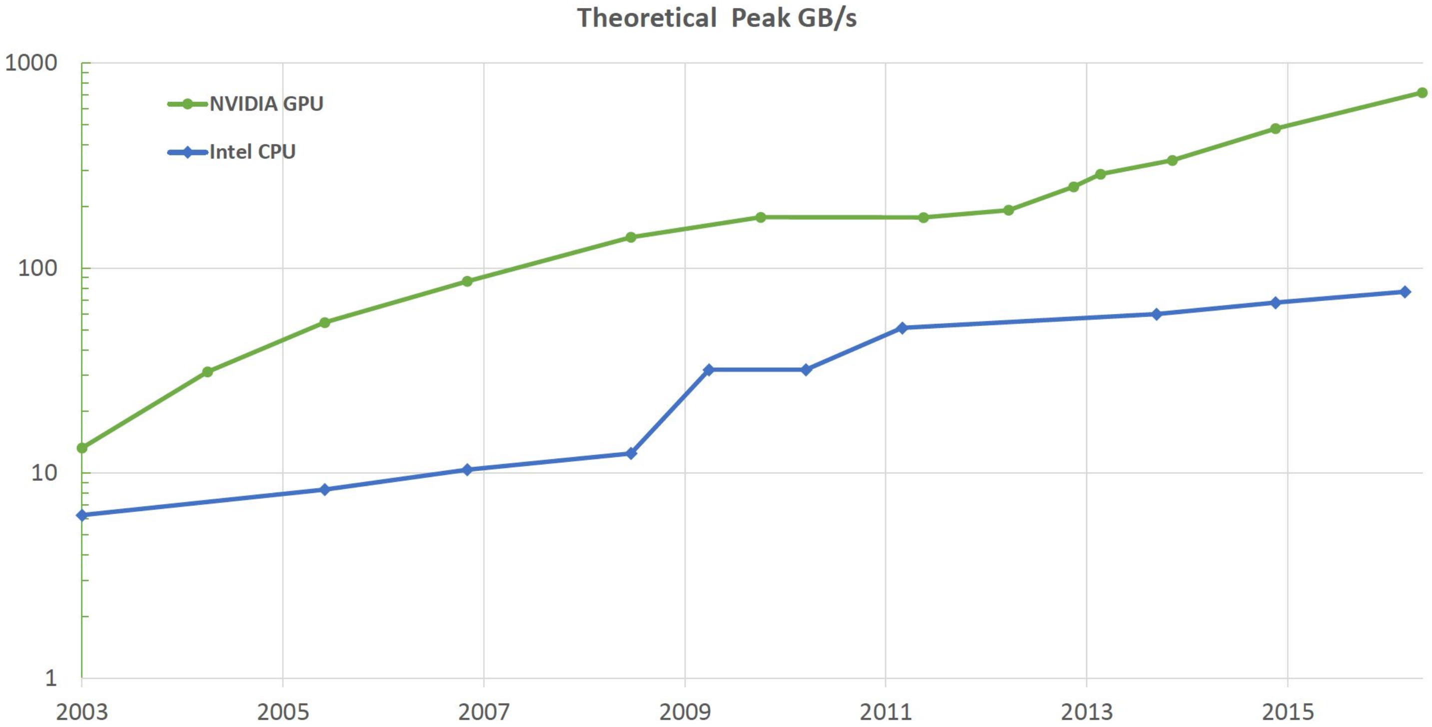
L2 cache: 200 cycles

DRAM: 350 cycles

(for Nvidia Maxwell architecture)

* http://lpgpu.org/wp/wp-content/uploads/2013/05/poster_andresch_acaces2014.pdf

Memory bandwidth comparison



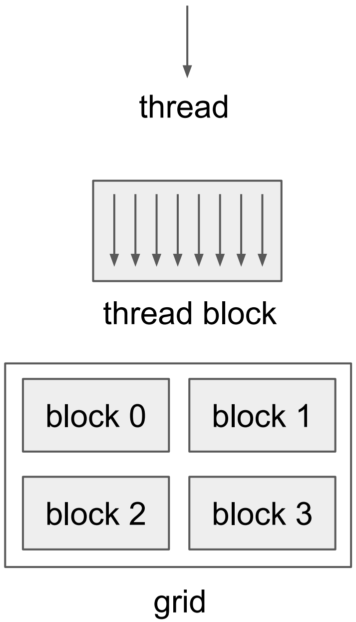
* https://github.com/oxford-cs-deepnlp-2017/lectures/blob/master/Lecture%206%20-%20Nvidia%20RNNs%20and%20GPUs.pdf

Nvidia GPU Comparison

GPU	Tesla K40 (2014)	Titan X (2015)	Titan X (2016)
Architecture	Kepler GK110	Maxwell GM200	Pascal GP102
Number of SMs	15	24	28
CUDA cores	2880 (192 * 15SM)	3072 (128 * 24SM)	3584 (128 * 28SM)
Max clock rate	875 MHz	1177 MHz	1531 MHz
FP32 GFLOPS	5040	7230	10970
32-bit Registers / SM	64K (256KB)	64K (256KB)	64K (256KB)
Shared Memory / SM	16 KB / 48 KB	96 KB	64 KB
L2 Cache / SM	1.5 MB	3 MB	3 MB
Global DRAM	12 GB	12 GB	12 GB
Power	235 W	250 W	250 W

CUDA Programming Model

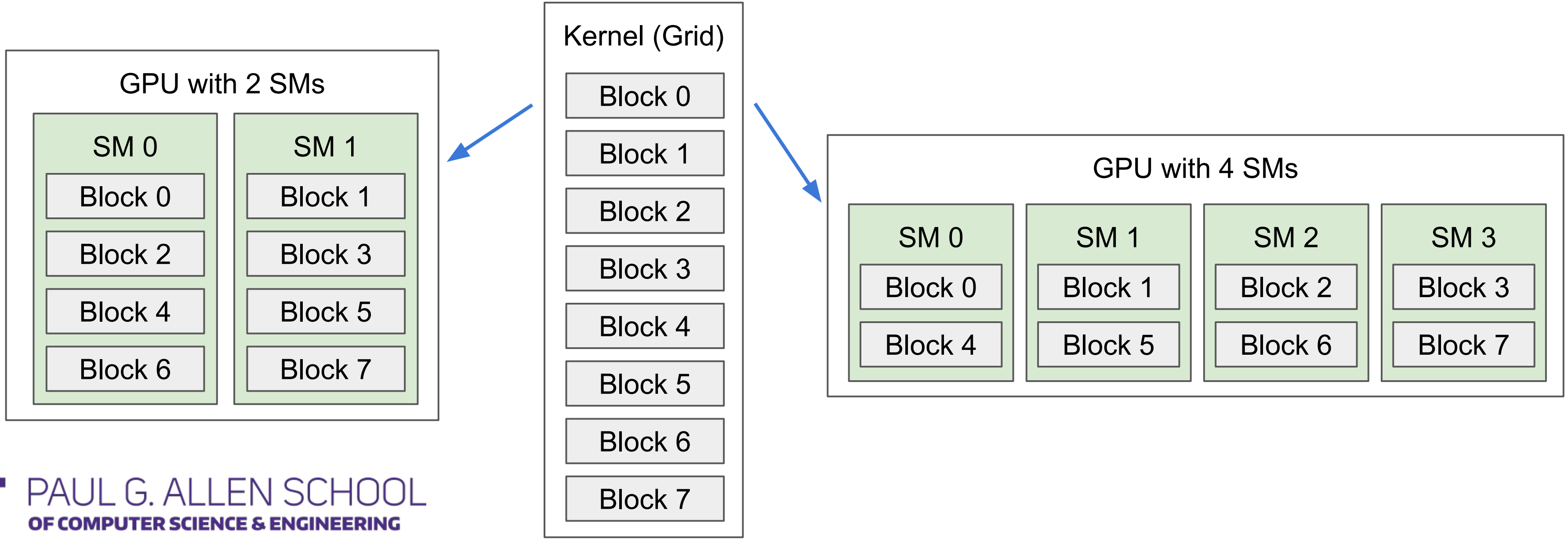
Thread Hierarchy



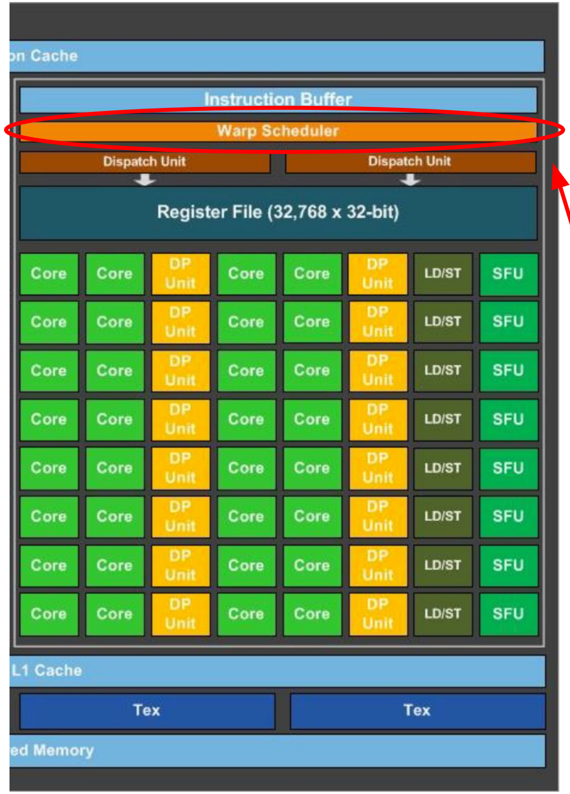
- Programmer writes code for a single thread in simple C program.
  - All threads executes the same code, but can take different paths.
- Threads are grouped into a block.
  - Threads within the same block can synchronize execution.
- Blocks are grouped into a grid.
  - Blocks are independently scheduled on the GPU, can be executed in any order.
- A kernel is executed as a grid of blocks of threads.

Kernel Execution

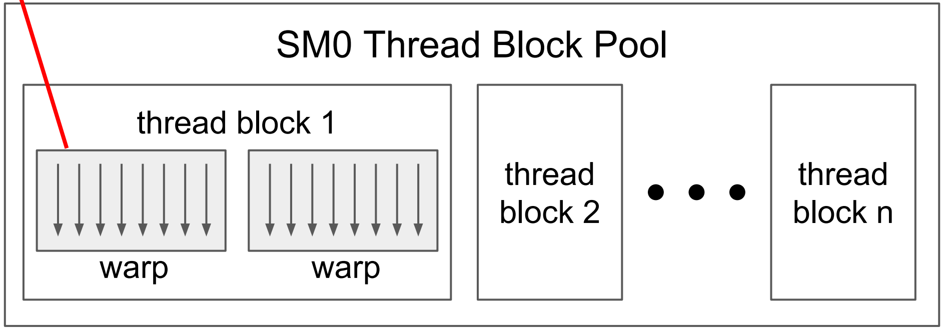
- Each block is executed by one SM and does not migrate.
- Several concurrent blocks can reside on one SM depending on block's memory requirement and the SM's memory resources.



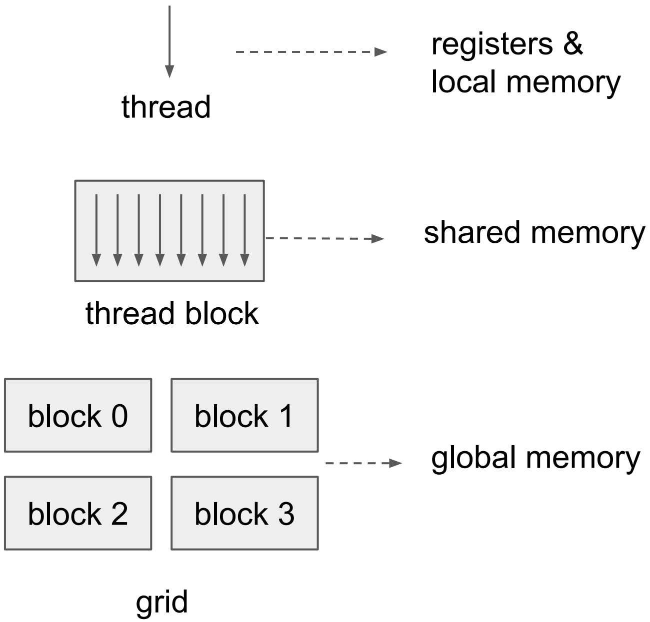
Kernel Execution

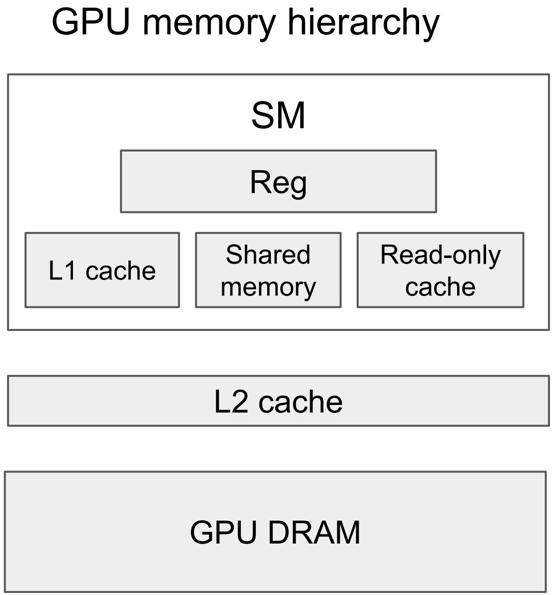


- A warp consists of 32 threads
  - A warp is the basic schedule unit in kernel execution.
- A thread block consists of 32-thread warps.
- Each cycle, a warp scheduler selects one ready warps and dispatches the warps to CUDA cores to execute.



Thread Hierarchy & Memory Hierarchy





Example: Vector Add

```
// compute vector sum C = A + B
Void vecAdd_cpu(const float* A, const float* B, float* C, int n) {
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}
```

Example: Vector Add

```
// compute vector sum C = A + B
Void vecAdd_cpu(const float* A, const float* B, float* C, int n) {
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}
```

```
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Example: Vector Add

global index	0	1	2	3	4	5	6	7	8	9	10	11
threadIdx.x	0	1	2	3	0	1	2	3	0	1	2	3
blockIdx.x	0				1				2			

Suppose each block only includes 4 threads: blockDim.x = 4

```
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Compute the global index

Example: Vector Add

global index	0	1	2	3	4	5	6	7	8	9	10	11
threadIdx.x	0	1	2	3	0	1	2	3	0	1	2	3
blockIdx.x	0				1				2			

Suppose each block only includes 4 threads: blockDim.x = 4

```
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Each thread only performs one pair-wise addition

Example: Vector Add (Host)

```
#define THREADS_PER_BLOCK   512
void vecAdd(const float* A, const float* B, float* C, int n) {
    float *d_A, *d_B, *d_C;
    int size = n * sizeof(float);
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);
    int nblocks = (n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    vecAddKernel<<<nblocks, THREADS_PER_BLOCK>>>(d_A, d_B, d_C, n);
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

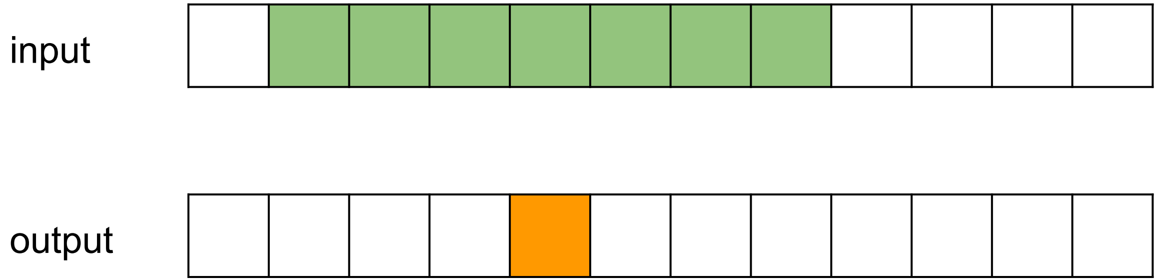
Example: Vector Add (Host)

```
#define THREADS_PER_BLOCK   512
void vecAdd(const float* A, const float* B, float* C, int n) {
    float *d_A, *d_B, *d_C;
    int size = n * sizeof(float);
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);
    int nblocks = (n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    vecAddKernel<<<nblocks, THREADS_PER_BLOCK>>>(d_A, d_B, d_C, n);
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Launch the GPU kernel asynchronously

Example: Sliding Window Sum

- Consider computing the sum of a sliding window over a vector
  - Each output element is the sum of input elements within a radius
  - Example: image blur kernel
- If radius is 3, each output element is sum of 7 input elements



A naive implementation

```
#define RADIUS 3
__global__ void windowSumNaiveKernel(const float* A, float* B, int n) {
    int out_index = blockDim.x * blockIdx.x + threadIdx.x;
    int in_index = out_index + RADIUS;
    if (out_index < n) {
        float sum = 0.;
        for (int i = -RADIUS; i <= RADIUS; ++i) {
            sum += A[in_index + i];
        }
        B[out_index] = sum;
    }
}
```

A naive implementation

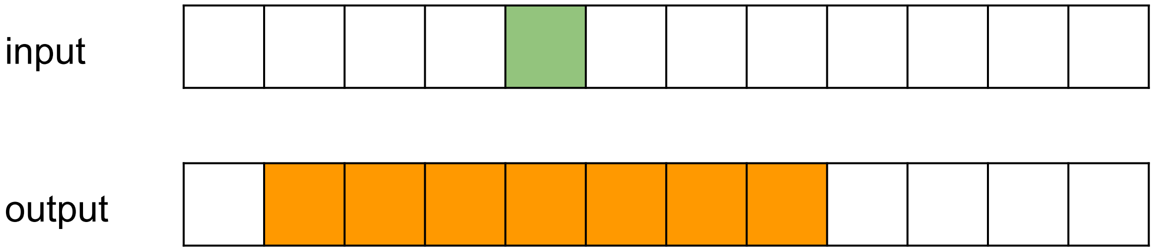
```
void windowSum(const float* A, float* B, int n) {
    float *d_A, *d_B;
    int size = n * sizeof(float);
    cudaMalloc((void **) &d_A, (n + 2 * RADIUS) * sizeof(float));
    cudaMemset(d_A, 0, (n + 2 * RADIUS) * sizeof(float));
    cudaMemcpy(d_A + RADIUS, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    dim3 threads(THREADS_PER_BLOCK, 1, 1);
    dim3 blocks((n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK, 1, 1);
    windowSumNaiveKernel<<<blocks, threads>>>(d_A, d_B, n);
    cudaMemcpy(B, d_B, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B);
}
```

How to improve it?

- For each element in the input, how many times it is loaded?

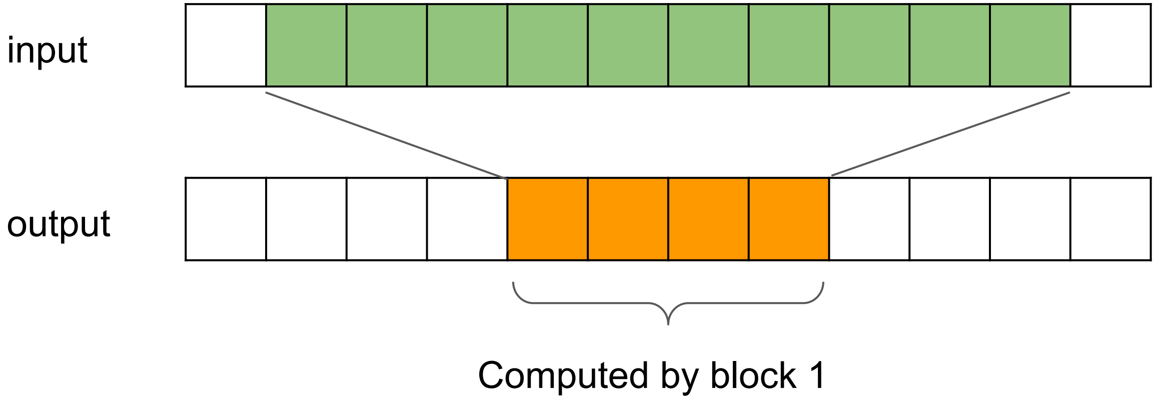
How to improve it?

- For each element in the input, how many times it is read?
  - Each input element is read 7 times!
  - Neighboring threads read most of the same elements
- How can we avoid redundant reading of data?



Sharing data between threads within a block

- A thread block first cooperatively loads the needed input data into the shared memory.



Kernel with shared memory

```
__global__ void windowSumKernel(const float* A, float* B, int n) {
    __shared__ float temp[THREADS_PER_BLOCK + 2 * RADIUS];
    int out_index = blockDim.x * blockIdx.x + threadIdx.x;
    int in_index = out_index + RADIUS;
    int local_index = threadIdx.x + RADIUS;
    if (out_index < n) {
        temp[local_index] = A[in_index];
        if (threadIdx.x < RADIUS) {
            temp[local_index - RADIUS] = A[in_index - RADIUS];
            temp[local_index + THREADS_PER_BLOCK] = A[in_index+THREADS_PER_BLOCK];
        }
        __syncthreads();
```

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

Kernel with shared memory

```
        float sum = 0.;
        for (int i = -RADIUS; i <= RADIUS; ++i) {
            sum += temp[local_index + i];
        }
        B[out_index] = sum;
    }
}
```

Performance comparison

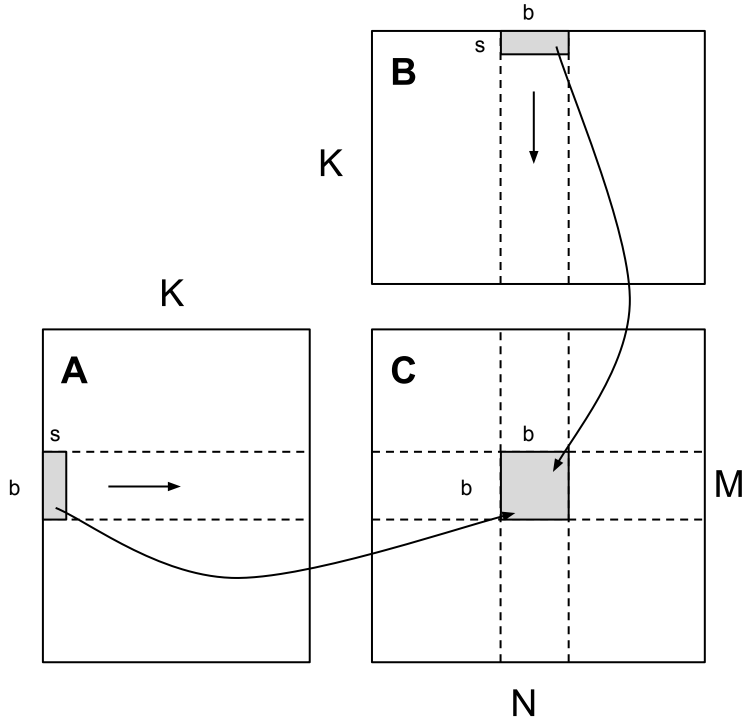
Demo!

Code:
https://github.com/dlsys-course/dlsys-course.github.io/blob/master/examples/window_sum.cu

Case study of efficient GPU kernels

Case study: GEMM

C = A x B
A: MxK matrix
B: KxN matrix
C: MxN matrix

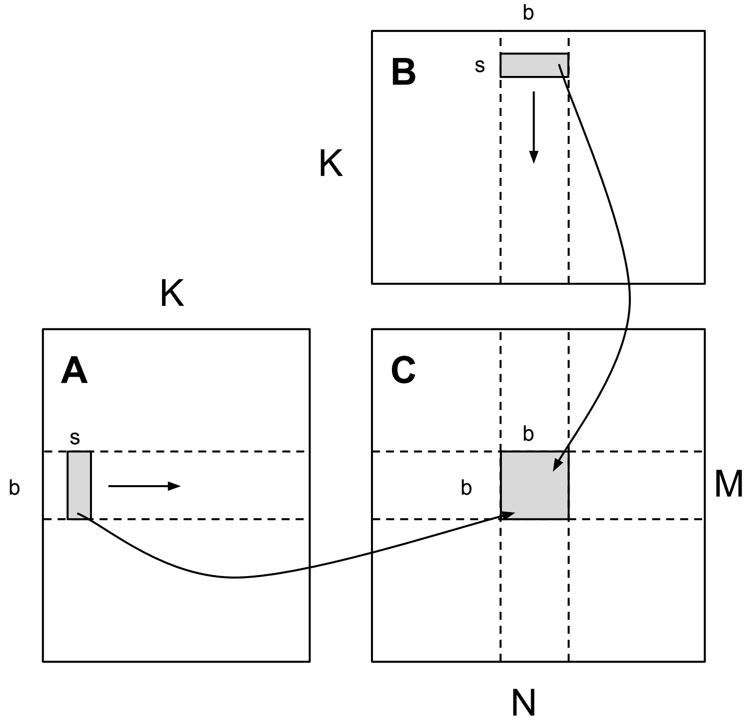


Workload of a thread block

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

Case study: GEMM

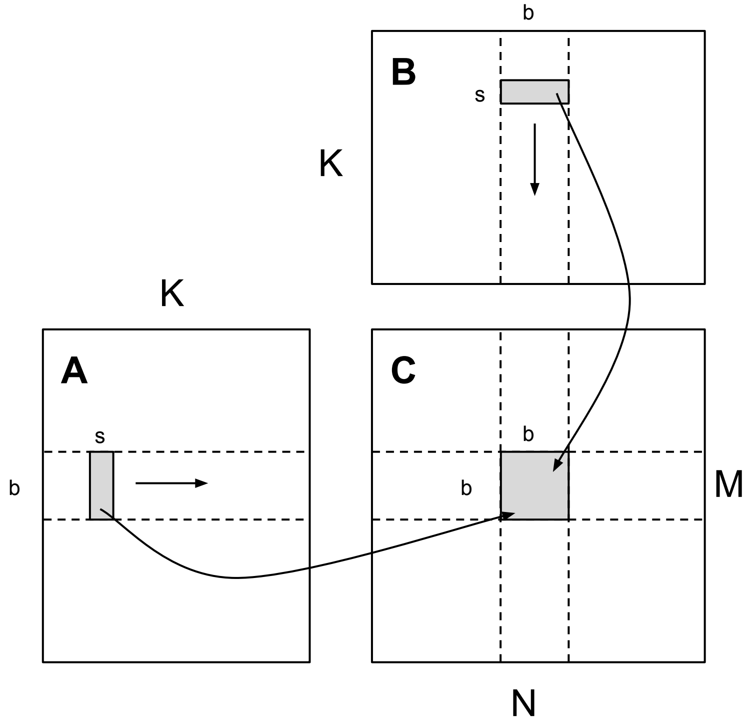
C = A x B
A: MxK matrix
B: KxN matrix
C: MxN matrix



Workload of a thread block

Case study: GEMM

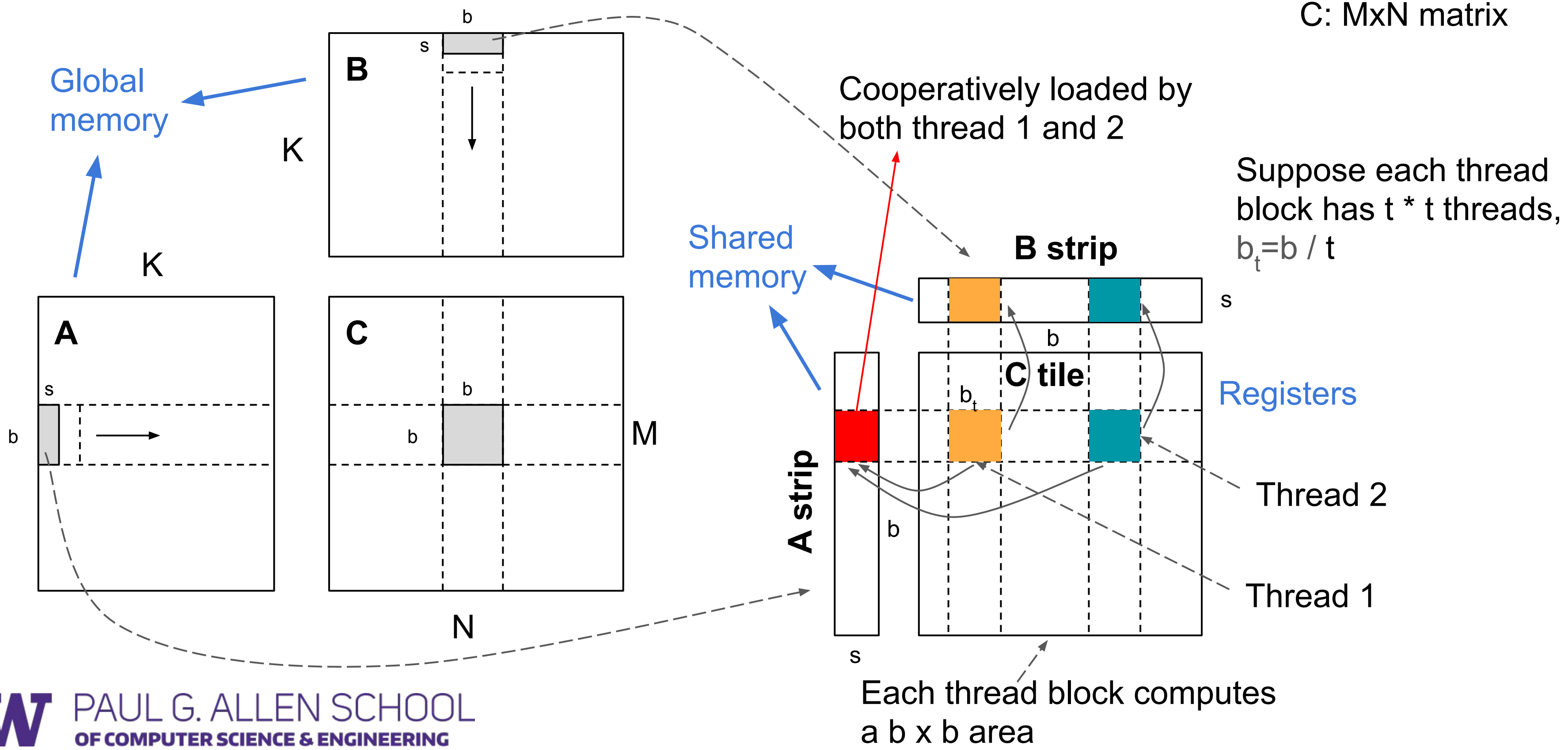
C = A x B
A: MxK matrix
B: KxN matrix
C: MxN matrix



Workload of a thread block

Case study: GEMM

C = A x B
A: MxK matrix
B: KxN matrix
C: MxN matrix



Case study: GEMM pseudocode

```
block_dim: <M / b, N / b>
thread_dim: <t, t>
// thread function
__global__ void SGEMM(float *A, float *B, float *C, int b, int s) {
  __shared__ float sA[2][b][s], sB[2][s][b]; // shared by a thread block
  float rC[b_t][b_t] = {0};                   // thread local buffer, in the registers
  Cooperative fetch first strip from A, B to sA[0], sB[0]
  __sync_threads();
  for (k = 0; k < K / s; k += 1) {
    Cooperative fetch next strip from A, B to sA[(k+1)%2], sB[(k+1)%2]
    __sync_threads();
    for (kk = 0; kk < s; kk += 1) {
      for (j = 0; j < b_t; j += 1) {         // unroll loop
         for (i = 0; i < b_t; i += 1) {      // unroll loop
           rC[j][i] += sA[k%2][threadIdx.x*b_t+j][kk]*sB[k%2][kk][threadIdx.y*b_t+i];
         }
  }}}
  Write rC back to C
}
```

Run in parallel

Case study: GEMM

More details see:

- http://homes.cs.washington.edu/~tws10/cse599i/CSE%20599%20I%20Accelerated%20Computing%20-%20Programming%20GPUs%20Lecture%204.pdf
- Lai, Junjie, and André Seznec. "Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs." Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on. IEEE, 2013.

Case study: Reduction Sum

http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Tips for high performance

- Use existing libraries, which are highly optimized, e.g. cublas, cudnn.
- Use nvprof or nvvp (visual profiler) to debug the performance.
- Use high level language to write GPU kernels.

References

- CUDA Programming Guide: http://docs.nvidia.com/cuda/cuda-c-programming-guide/