High Performance Computing for Science and Engineering II

29.4.2019 - **Lecture 9: CUDA Optimization**

Lecturer: Dr. Sergio Martin

CSElab

Computational Science & Engineering Laboratory

ETH zürich

General Info

Semester Exam

- **When:** May 27th – 10.00am – 1.00pm
- **Where:** HG E7
- **What**: 3 Hours - Handwritten Only
- Can contain code-related questions.
- We'll let you know what aids you can bring.

Next Week’s Exercise Session

- We will have another interactive session (CUDA)
- Bring your laptop.

Today's Class

A brief history of memory hierarchies.

- From first cache structures modern CPUs

CUDA: Optimizing Memory Access

- Shared memory, register memory. Example.

Kernel Decomposition

- Vector reduction pattern.

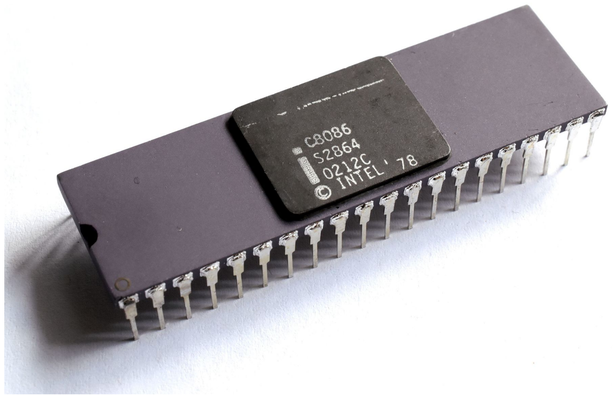
Homework 5

- Goals and Grading.

Hardware Hierarchies: A brief History

The Intel 8086 Processor

1978 - Intel Releases 8086, the first 16-bit processor of the x86 architecture.



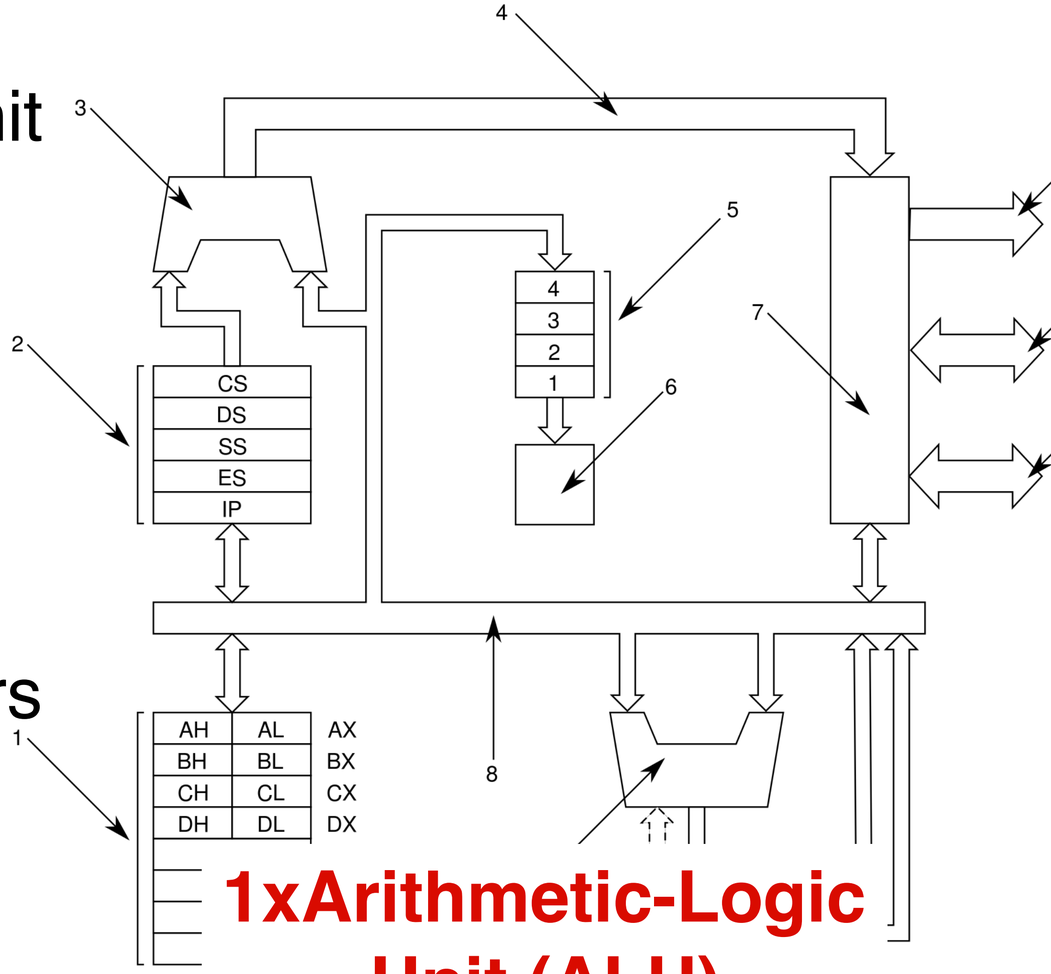
29k Transistors - 5Mhz

16-bit = 64k
Max Memory Address

Address Calculator Unit

Segment Selectors
(16-bit)

General-Purpose Registers
(16-bit)



1xArithmetic-Logic Unit (ALU)

Further Read : Computer Organization and Design: The Hardware/Software Interface

Picture : https://commons.wikimedia.org/wiki/File:Intel_8086_block_scheme.svg

(32-bit) Extended Registers

1986 - Intel Releases 80386, its first 32-bit Processor.

275k Transistors - 20Mhz

32-bit = 4gb
Max Memory Address

Extended Registers

Applications Operate on Larger Data Sets

Additional Pressure on RAM

Intel 80386 registers

31 ... 15 ... 07 ... 00 *(bit position)*			
Main registers *(8/16/32 bits)*			
EAX	AX	AL	Accumulator register
EBX	BX	BL	Base register
ECX	CX	CL	Count register
EDX	DX	DL	Data register
Index registers *(16/32 bits)*			
ESI	SI		Source Index
EDI	DI		Destination Index
EBP	BP		Base Pointer
ESP	SP		Stack Pointer
Program counter *(16/32 bits)*			
EIP	IP		Instruction Pointer
Segment selectors *(16 bits)*			
	CS		Code Segment
	DS		Data Segment
	ES		Extra Segment
	FS		F Segment
	GS		G Segment
	SS		Stack Segment

Status register

17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00 *(bit position)*																		
V	R	0	N	IOPL		O	D	I	T	S	Z	0	A	0	P	1	C	EFlags

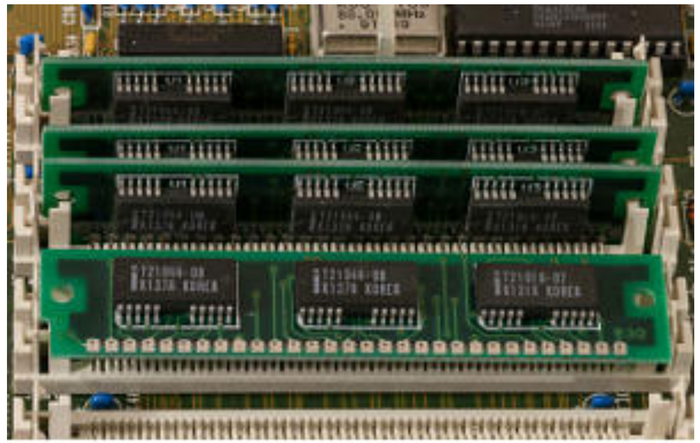
Picture Source: CPU collection Konstantin Lanzet

CPU/RAM Latency

275k Transistors - 20Mhz
GP Register Capacity: 16 Bytes
Register Memory Latency: 2-5ns

Latency Ratio: 24x

Capacity Ratio: ~10^6

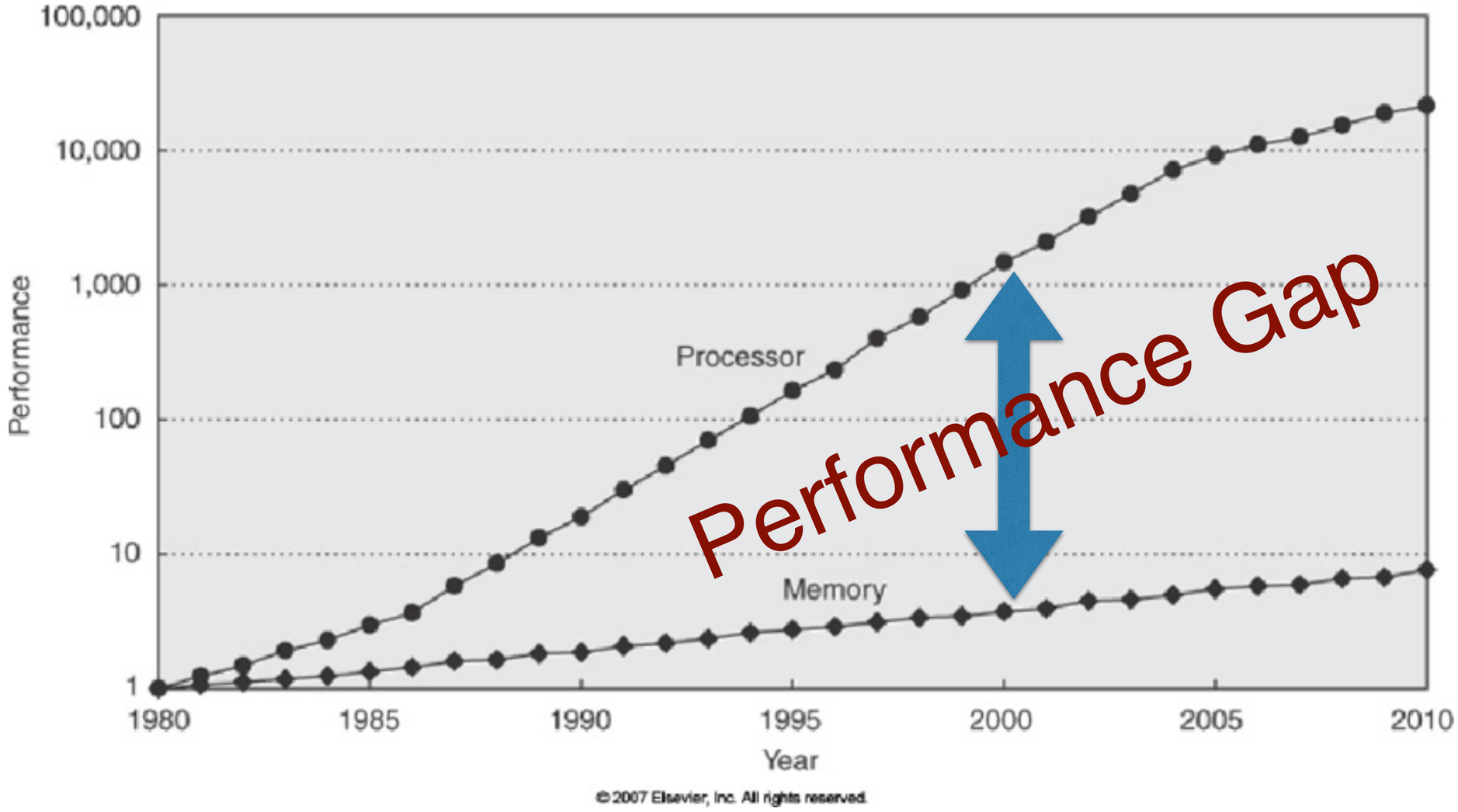


Asynchronous DRAM
Capacity: 1-64 Mbytes
Latency: ~120ns

Picture Source: CPU collection Konstantin Lanzet

Memory Performance Gap

Problem - Growing disparity between register and RAM latencies.



Picture Source: Computer Architecture: A Quantitative Approach Book by David A Patterson and John L. Hennessy

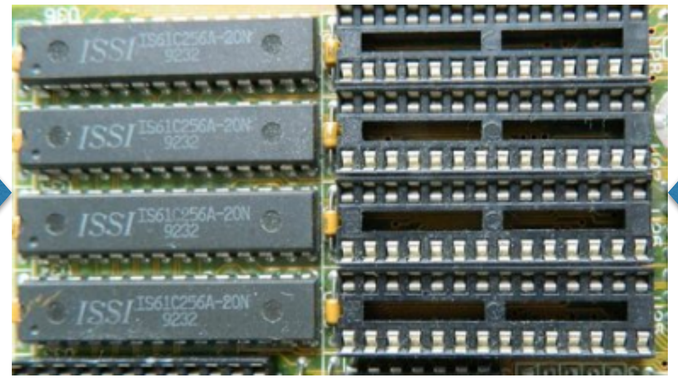
Cache Memory

1988 - Intel Releases 80386SX, the first commercial CPU with a Data-Cache Memory

Latency Ratio: 4x
Capacity Ratio: ~10^4

Latency Ratio: 6x
Capacity Ratio: ~10^2

275k Transistors - 20Mhz
GP Register Capacity: 16 Bytes
Register Memory Latency: 2-5ns



External Cache (SRAM)
Capacity: ~128 kbytes
Latency: 10-25ns

Asynchronous DRAM
Capacity: 1-64 Mbytes
Latency: ~120ns

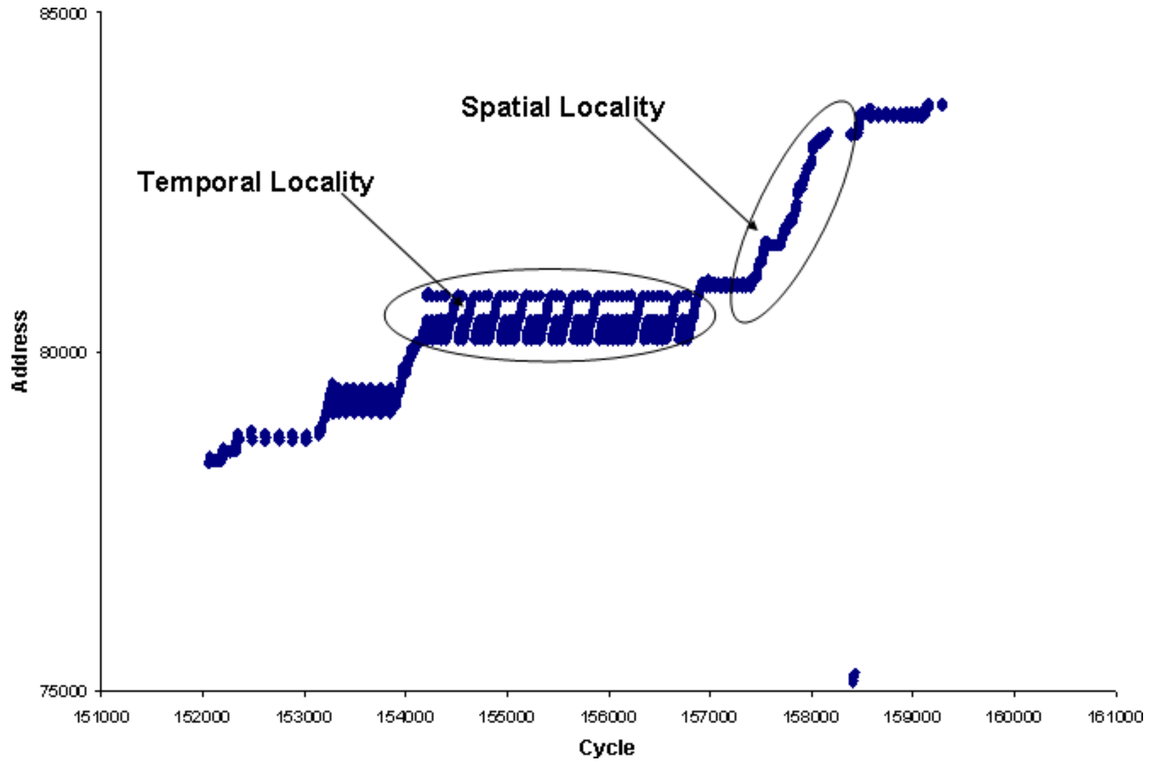
- Cache Memories do not improve the RAM->CPU Latency.
- Instead, they speed-up the reuse of data based on temporal and cache locality.

RAM is itself a cache for HDD!

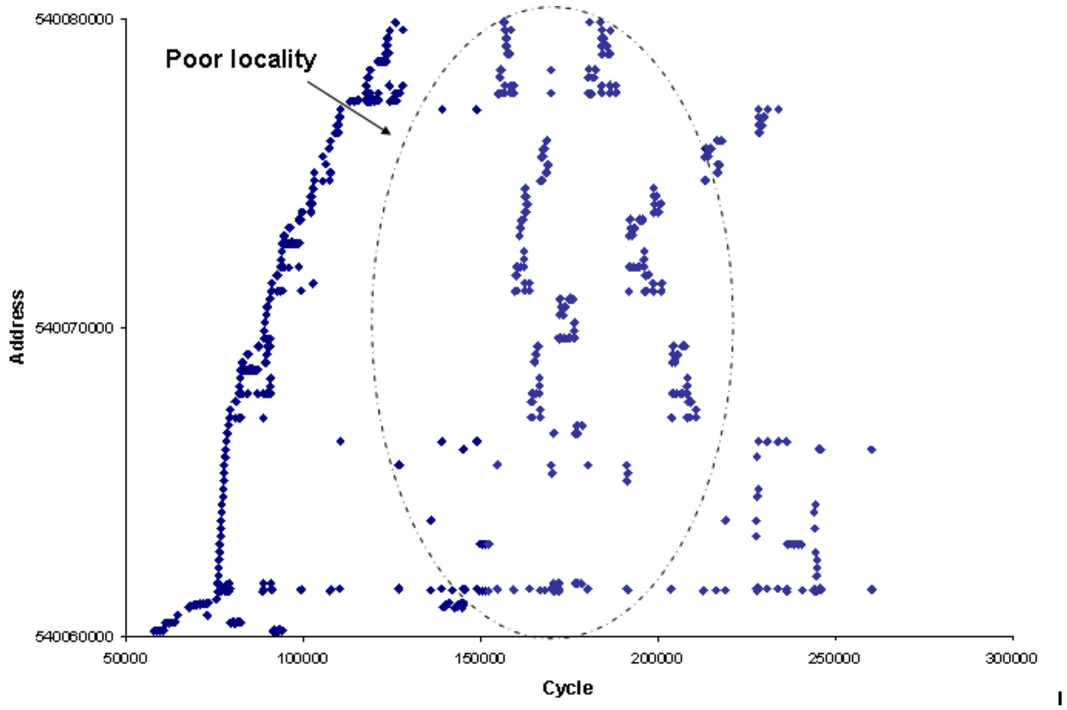
Picture Source: CPU collection Konstantin Lanzet

Cache Locality

Cache structures in modern processors benefit from both **Temporal** and **Spatial** locality.



High Cache Line Reuse



Frequent Cache Fails

Image source: Optimizing for instruction caches, part 1, Amir Kleen, Livadariu Mircea, Itay Peled, Erez Steinberg, Moshe Anschel, Freescale

Cache Hierarchy

201x - Modern Processors Employ Multiple Cache Levels

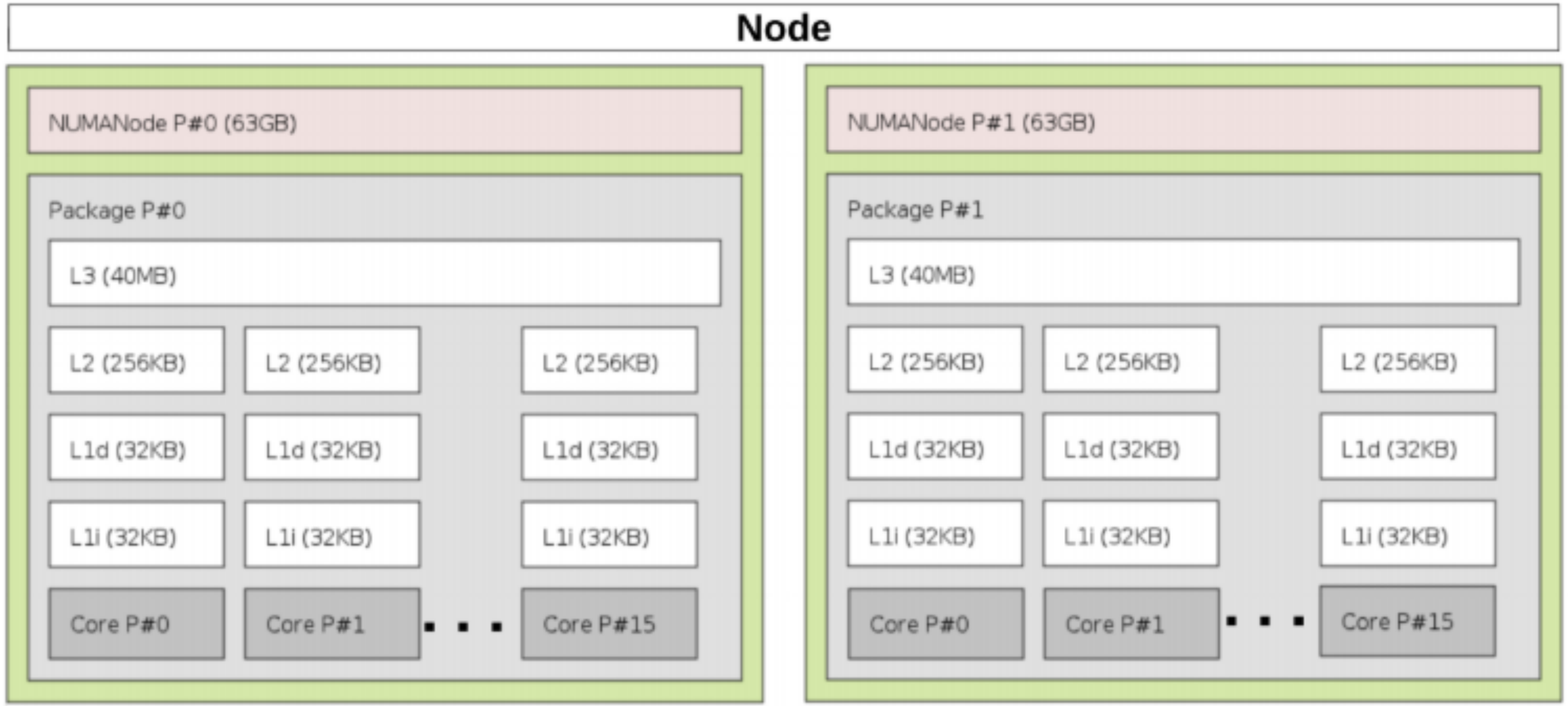
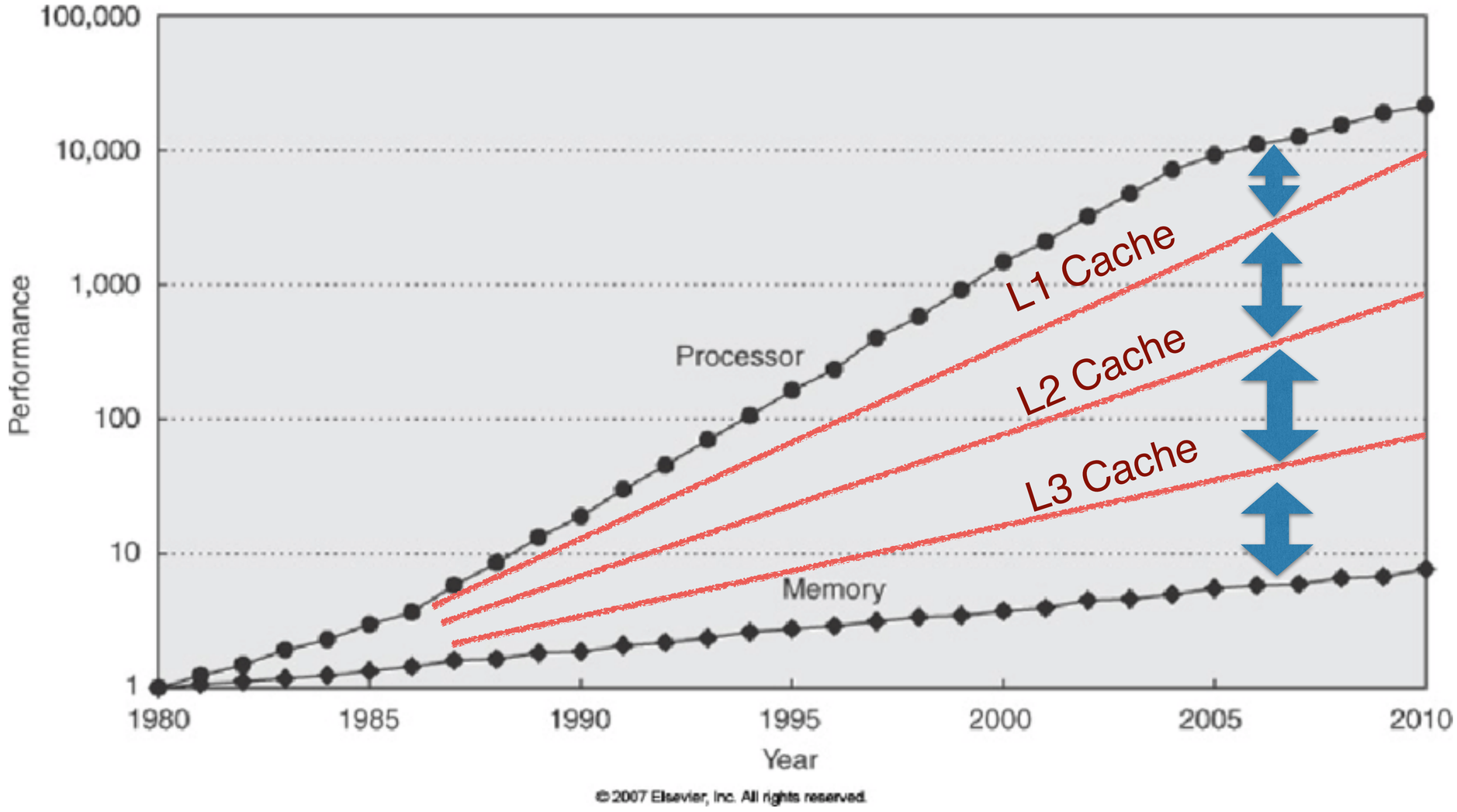


Figure 2.1: Configuration of a 32-core NERSC Cori Phase I (Haswell) Node.

RAM Latency

Solution - Cache memories bridges the gap between CPU and RAM performance

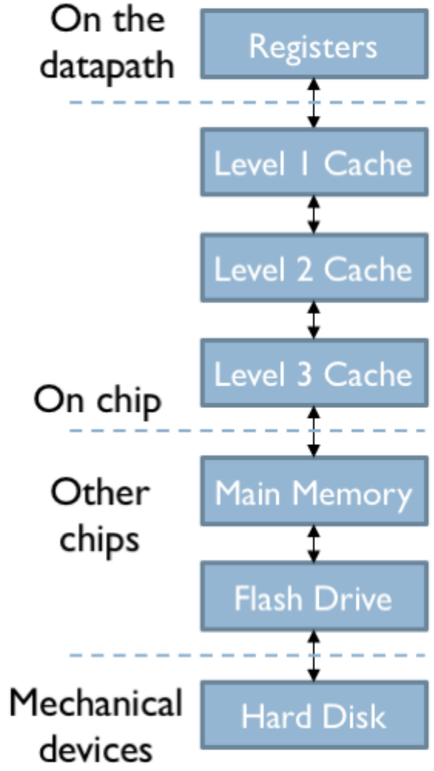


Picture Source: Computer Architecture: A Quantitative Approach Book by David A Patterson and John L. Hennessy

Memory Hierarchy

A Typical Memory Hierarchy

- Everything is a cache for something else…



Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
2-4 cycles	32 KB	Hardware
10 cycles	256 KB	Hardware
40 cycles	10 MB	Hardware
200 cycles	10 GB	Software/OS
10-100us	100 GB	Software/OS
10ms	1 TB	Software/OS

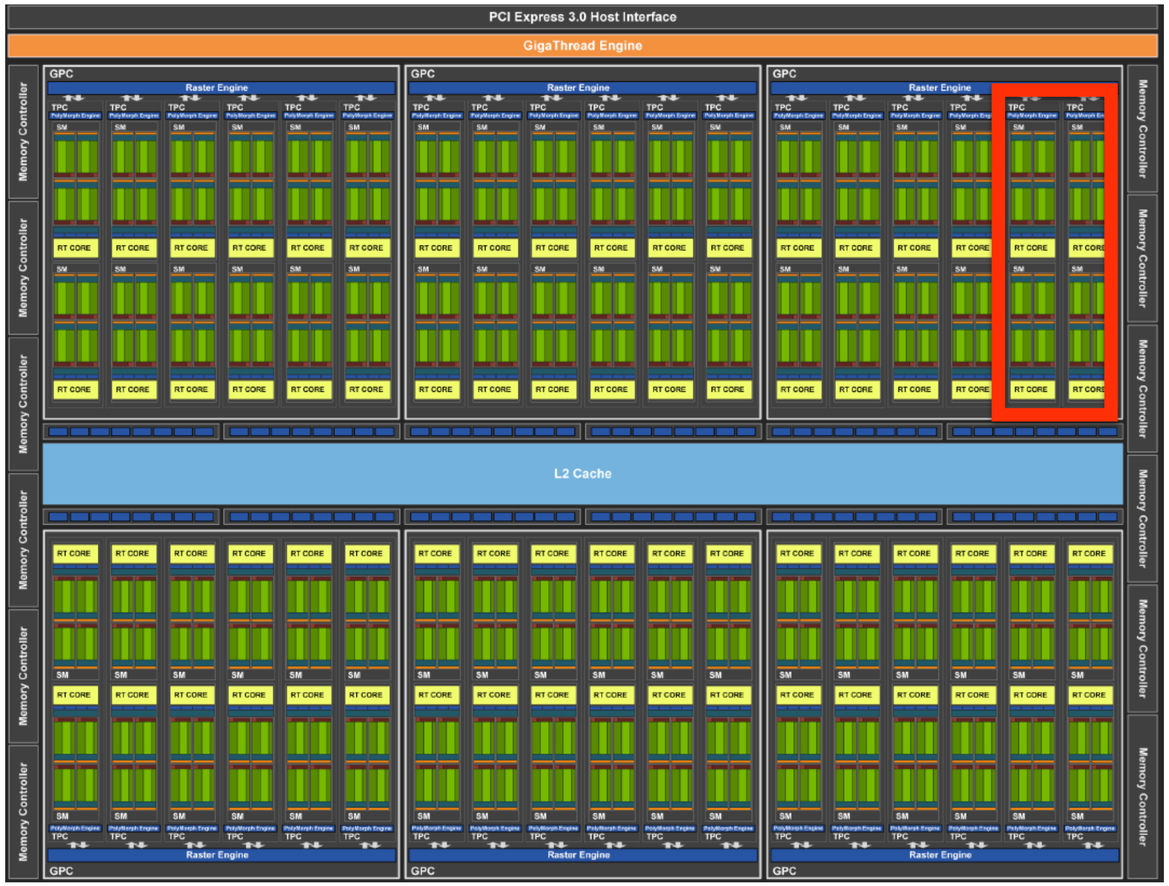
What does this mean?

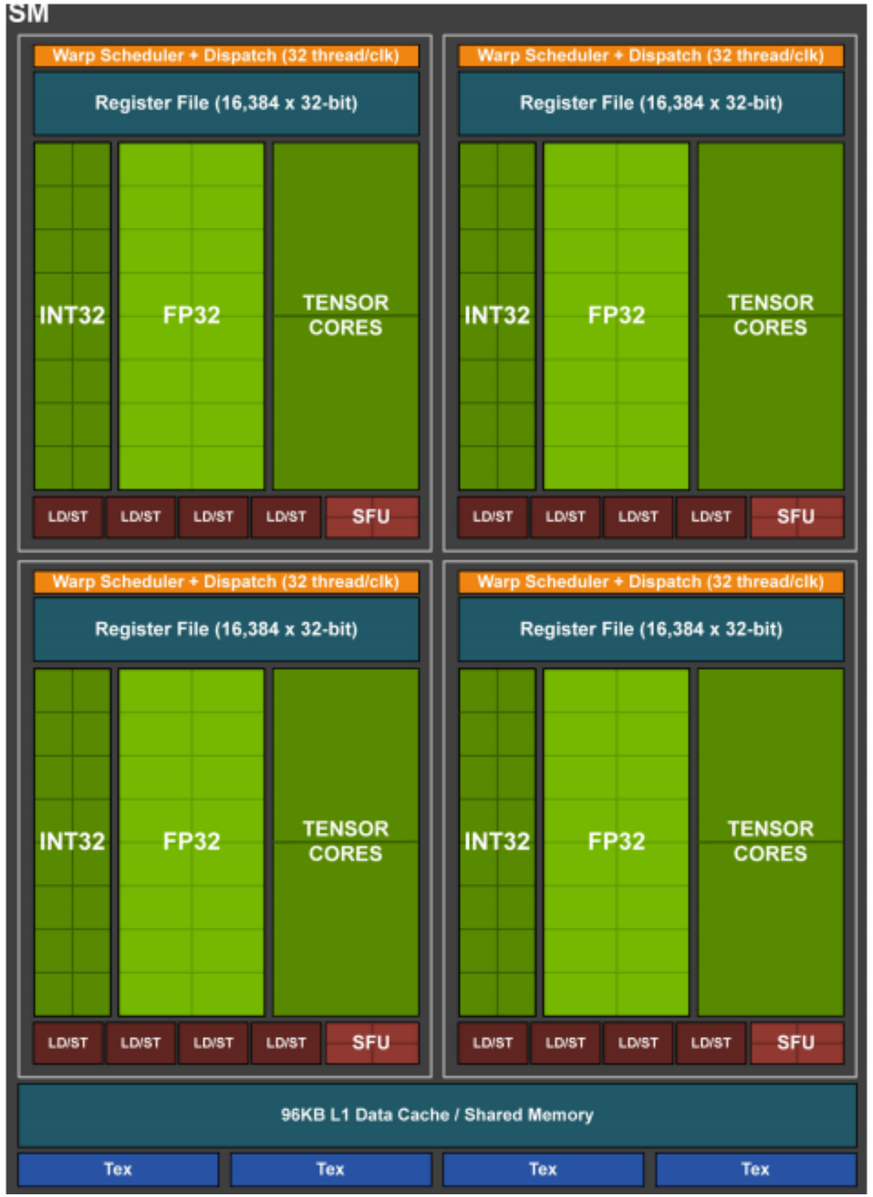
Source & Further Read : https://computationstructures.org/lectures/caches/caches.html

Memory Hierarchy on GPUs

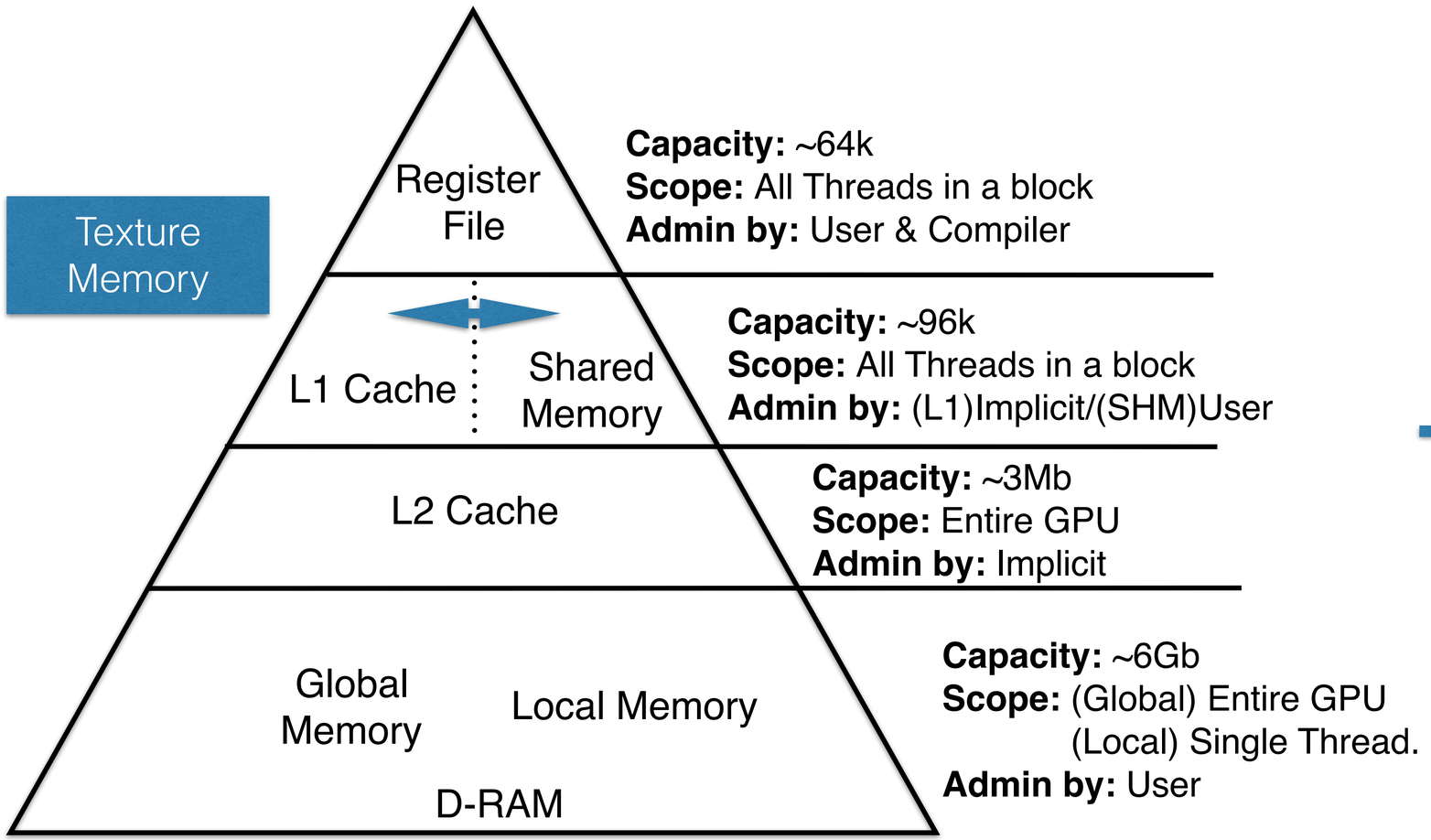
Turing TU102 Streaming Multiprocessor(SM)

Up to 4608 threads executing concurrently.





GPU Memory Hierarchy



Capacity: ~64k
Scope: All Threads in a block
Admin by: User & Compiler

Capacity: ~96k
Scope: All Threads in a block
Admin by: (L1)Implicit/(SHM)User

Capacity: ~3Mb
Scope: Entire GPU
Admin by: Implicit

Capacity: ~6Gb
Scope: (Global) Entire GPU
(Local) Single Thread.
Admin by: User

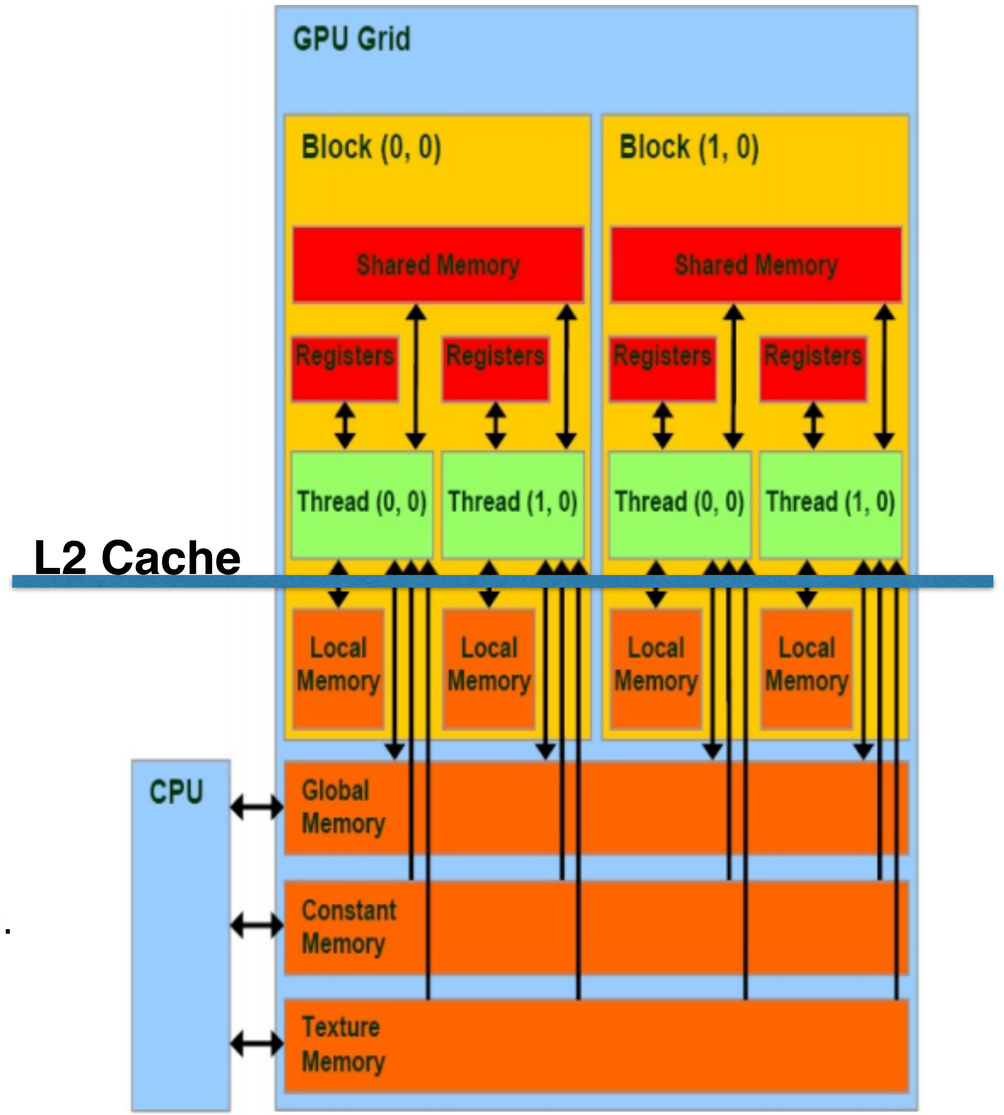


Image Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

GPU Memory Hierarchy

Variable declaration	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

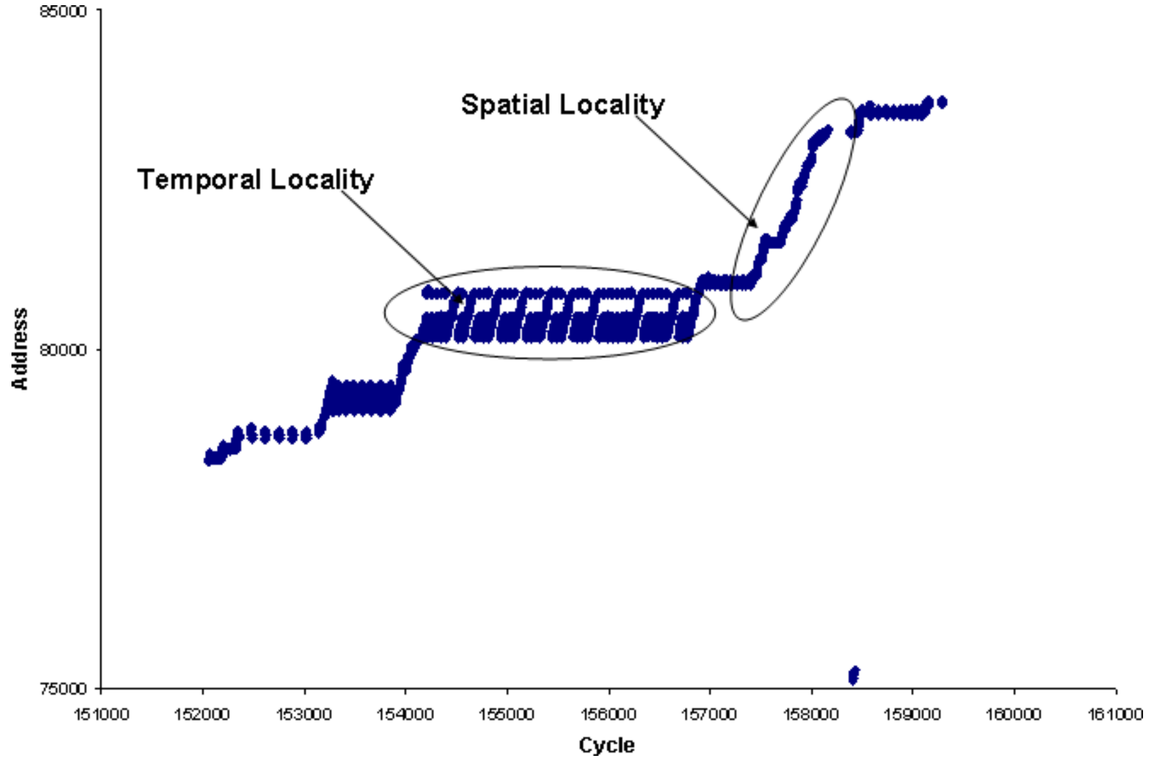
- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`

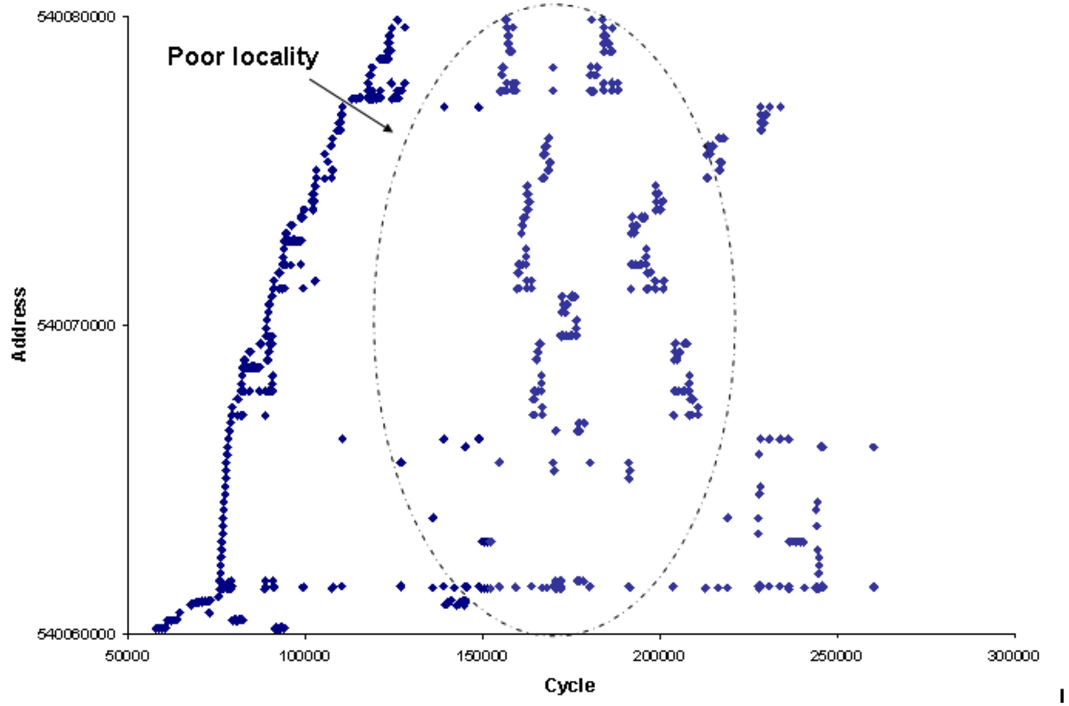
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory
  - Thread-local memory and spilled automatic variables is allocated in global memory

Source: David Kirk and Wen-Mei Hwu

Cache Locality

Cache structures in modern processors benefit from both **Temporal** and **Spatial** locality.





High Cache Line Reuse

Frequent Cache Fails

>> The caching/eviction decision is handled **implicitly** by the CPU guided by these heuristics <<

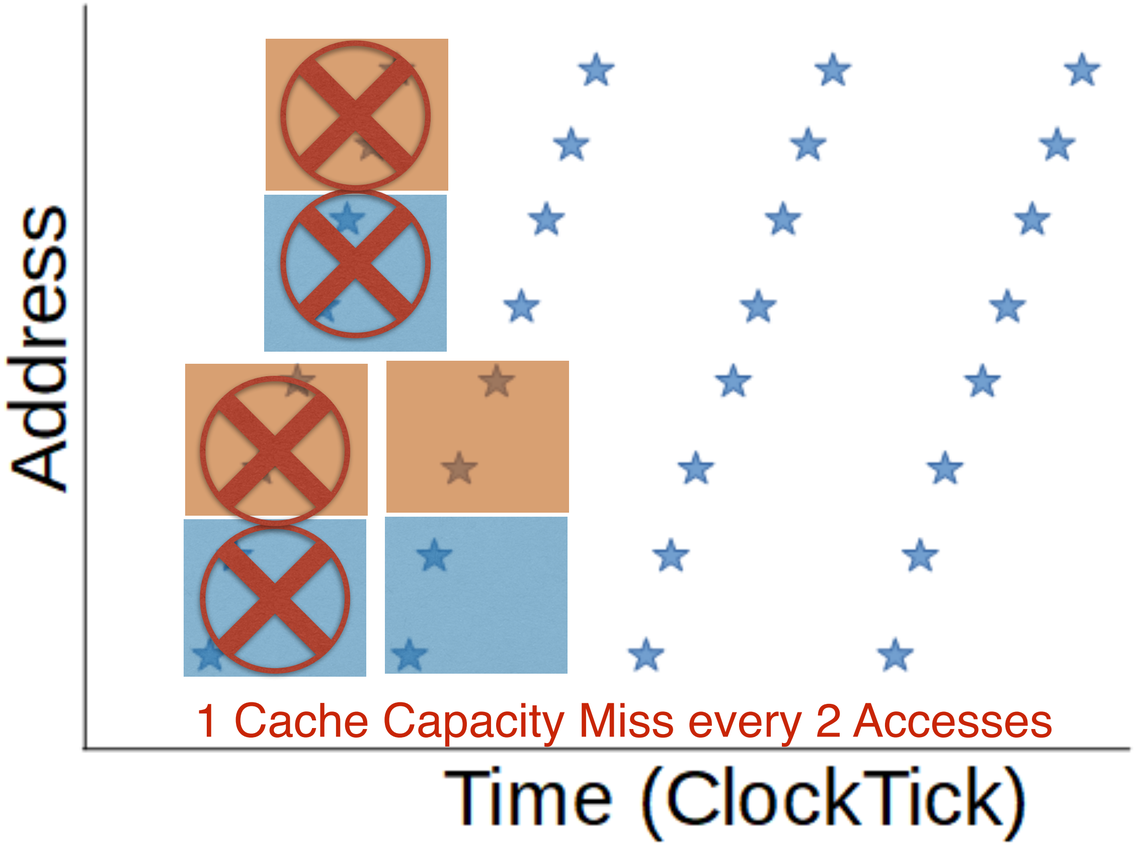
Dictates how a program has to be developed to harness its potential

This works well in the vast majority of cases, but may not always be the **optimal** strategy.

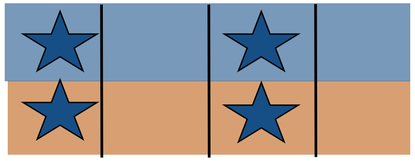
Image source: Optimizing for instruction caches, part 1, Amir Kleen, Livadariu Mircea, Itay Peled, Erez Steinberg, Moshe Anschel, Freescale

Cache Locality

Consider the following case: CPU with 2 Cache Lines Available - 4 Elements per Line



Cache Line Contents



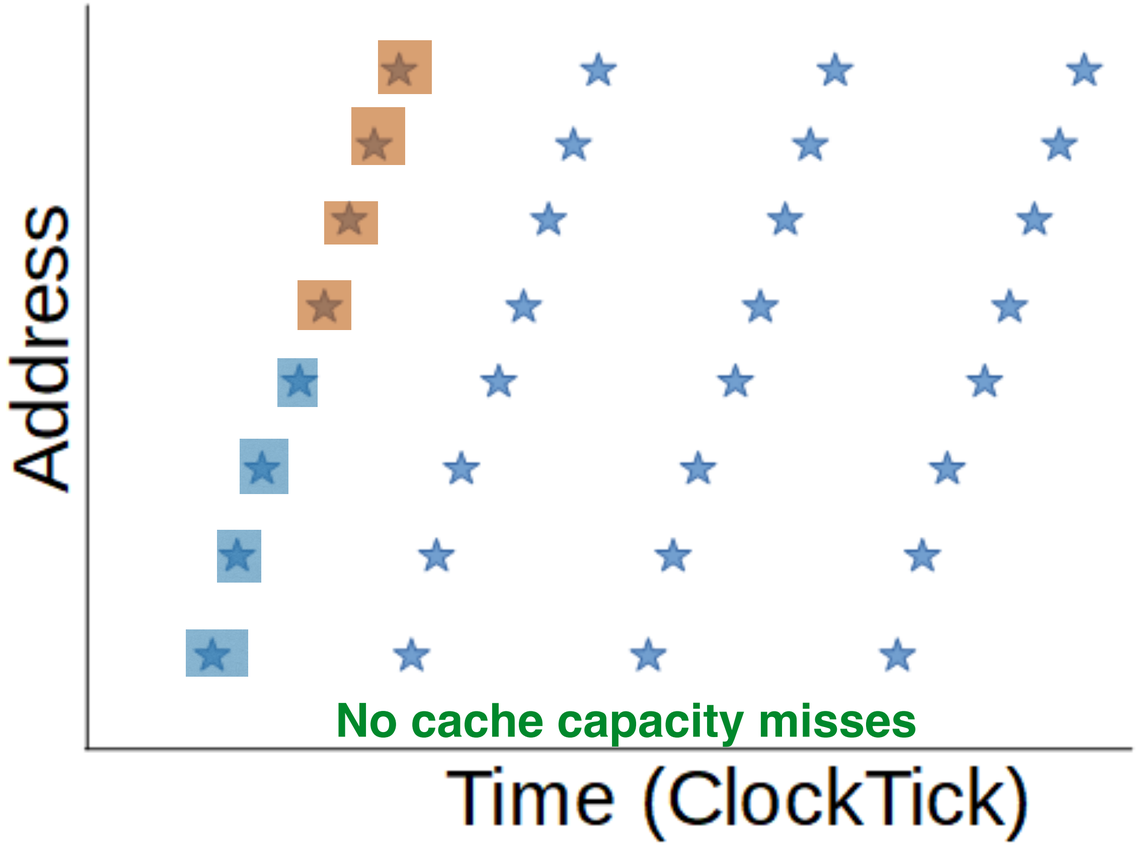
50% Efficiency

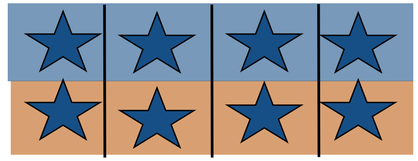
Explicit Cache

The user decides what data is loaded into cache -> Manual Prefetching.

Pseudo-Example:

```
for (int i = 0; i < N; i++)
cache[i] = prefetch(buf[i*2])
```





100% Efficiency

GPU Shared Memory

Key Points:

- Explicitly allocated/accessed by the programmer.
- Shared Among all threads in a block --> Requires proper synchronization.
- Remaining (unallocated) segments used for Local Memory and then L1 cache .
- Tailored to fit the application (not necessarily otherwise, like CPU cache)

Allocation:

- Host code: On Kernel Launch

```
size_t shmSize = 1000*sizeof(double); // The rest goes to L1 Cache
myKernel<<<grid, block, shmSize>>>(pars);
```

Shared Memory Size should not exceed the L1-cache size of your GPU

Usage:

- Device code: Inside the kernel

```
extern __shared__ double mySharedArray[];
s[i] = globalArray[i*n]; // Prefetching global memory into shared memory
```

Register / Local Memory

Register Optimization

- Registers are explicitly allocated/accessed by the programmer.

CPU:
Predefined by Hardware
through register names
Employed by the compiler
RAX ... RBX ...RCX

GPU:
General Space (64kb) for entire Block
Programmer defines and names registers:
double mySum = 0.0;
double myAverage = 0.0;

- GPU Compiler is less aggressive with optimizations due assumed parallelism
- Limited number of registers:
  NRegs * NThreads should not exceed Register File Size (e.g., 64kb)

  Warning: Excessive Registers will spill to local memory (DRAM)

- Exceptions:
  - Pre-defined arrays (e.g., myArray[512]) are stored in local memory.
  - Constant variables (e.g., const myPI = 3.14) may not count (are optimized out)

GPU Shared Memory (Static)

Example: 2D Problem - 4k Elements per side

```
#define BLOCKSIZE 32
N = 4096;
dim3 blockDim(BLOCKSIZE,BLOCKSIZE) // 512 Threads
dim3 gridDim(n/BLOCKSIZE, n/BLOCKSIZE) //  128x128 Blocks = 16k Blocks
my2dProblem<<<gridDim, blockDim>>>(grid1, grid2, N);
```

Device Code:

```
__global__ void my2DProblem(double* grid1, double* grid2, size_t N)
{
 const size_t m     = blockIdx.x * blockDim.x + threadIdx.x;
 const size_t t     = threadIdx.x;

 __shared__ double s0[BLOCKSIZE][BLOCKSIZE]; //  32*32*8 bytes = 8k bytes
 __shared__ double s1[BLOCKSIZE][BLOCKSIZE]; //  32*32*8 bytes = 8k bytes

 s[2*t + 0] = grid1[m];
 s[2*t + 1] = grid2[m];

 __syncthreads(); // Important: synchronize to make sure every thread in the block saved their values before accessing.
 double myCalc = 0.0;
 for (size_t i = 0; i < blockDim.x; i++) myCalc += sqrt(s0[i] + s1[i]);
}
```

GPU Shared Memory (Dynamic)

Example: 2D Problem - 4k Elements per side

```
N = 4096;
dim3 blockDim(32,32) // 512 Threads
dim3 gridDim(n/32, n/32) //  128x128 Blocks = 16k Blocks
size_t shmSize = 2*32*32*sizeof(double);
my2dProblem<<<gridDim, blockDim, shmSize>>>(grid1, grid2, N);
```

Shared Memory:

2 Doubles per Thread = 512*2*8bytes = 8kb

Device Code:

```
__global__ void my2DProblem(double* grid1, double* grid2, size_t N)
{
 const size_t m     = blockIdx.x * blockDim.x + threadIdx.x;
 const size_t t     = threadIdx.x;

 extern __shared__ double s[];
 s[2*t + 0] = grid1[m];
 s[2*t + 1] = grid2[m];

 __syncthreads(); // Important: synchronize to make sure every thread in the block saved their values before accessing.
 double myCalc = 0.0;
 for (size_t i = 0; i < blockDim.x; i++) myCalc += sqrt(s[2*i] + s[2*i+1]);
}
```

Register Memory:

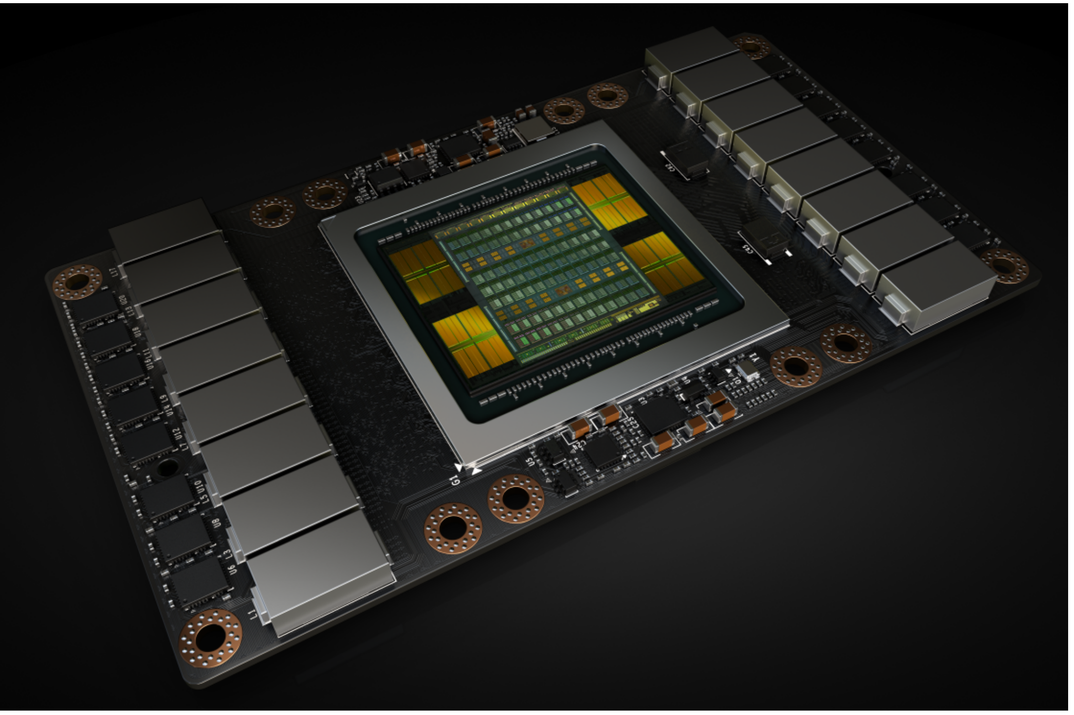
7 Registers * 512 Threads * 8bytes = 28kb

No spillage to local memory.

Banked Access to Global & Shared Memory

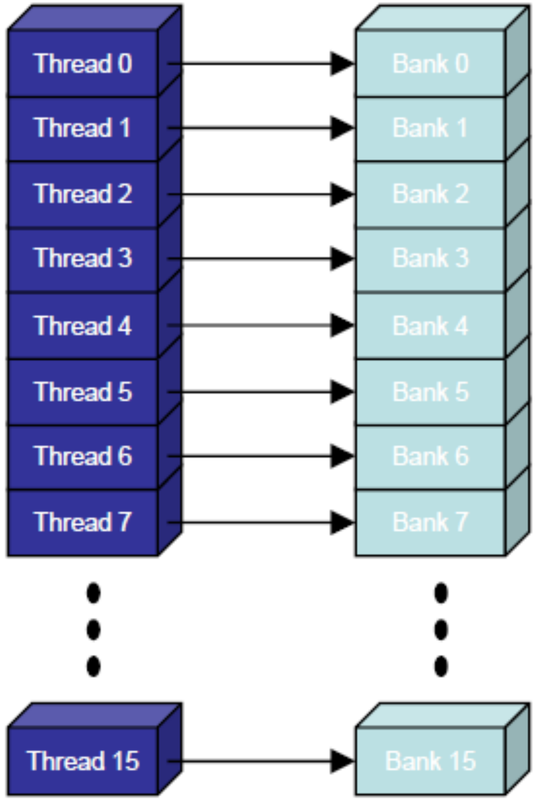
GPU RAM

Optimized for High-bandwidth Multi-banked access.

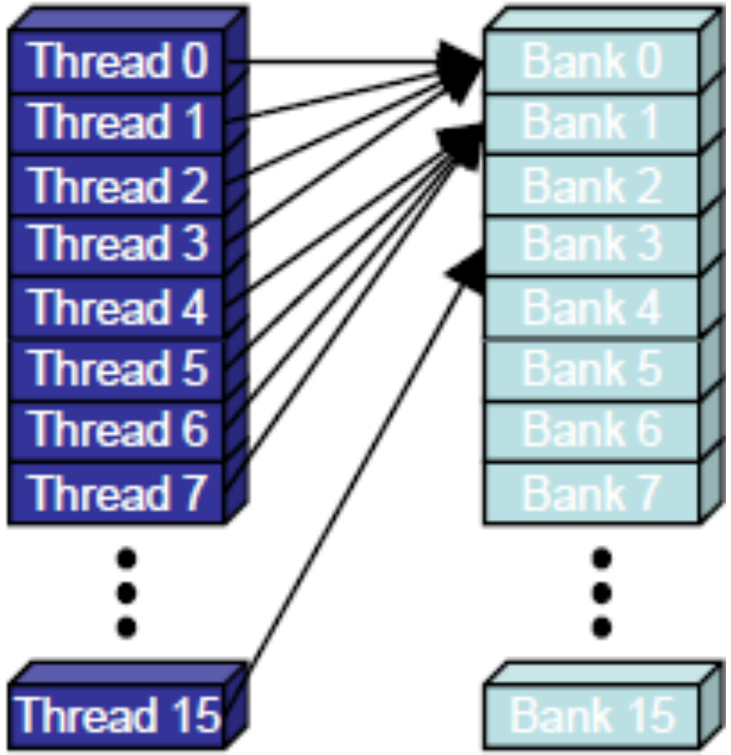


Source: NVIDIA

No Conflict



Multiple Conflicts



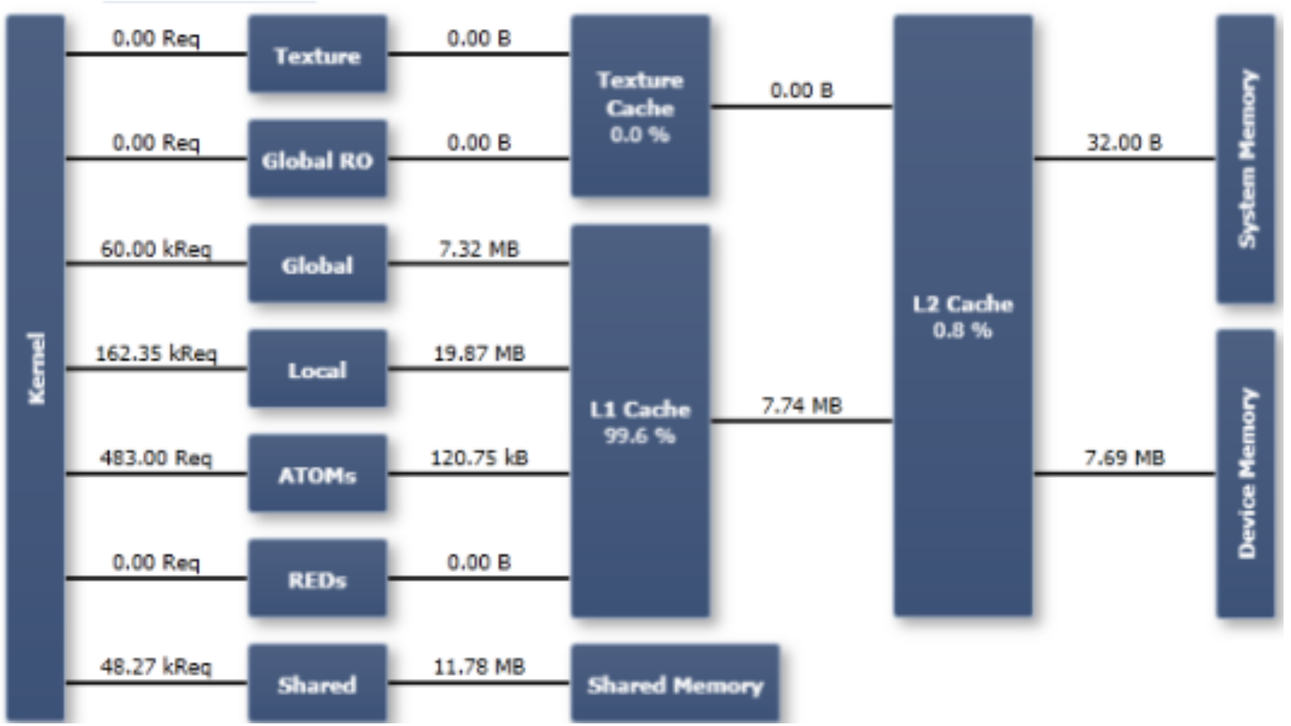
Source: CUDA Programming Blog

Profiling

NVIDIA Nsight - Displays the achieved (compute) occupancy and memory behavior of each kernel.

Kernel: kernelHitRegisterLimit **Grid Dim:** {240, 1, 1} 240 **Block Dim:** {512, 1, 1} 5...
Device: Quadro K6000 **Compute Capability:** 3.5 **Dyn Shm/Block:** 0 **Stat Shm/Block:** 0

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		3	16
Active Warps	44.81	48	64
Active Threads		1536	2048
Occupancy	70.01 %	75.00 %	100.00 %
Warps			
Threads/Block		512	1024
Warps/Block		16	32
Block Limit		4	16
Registers			
Registers/Thread		33	255
Registers/Block		20480	65536
Block Limit		3	16
Shared Memory			
Shared Memory/Block		0	49152
Block Limit		∞	16



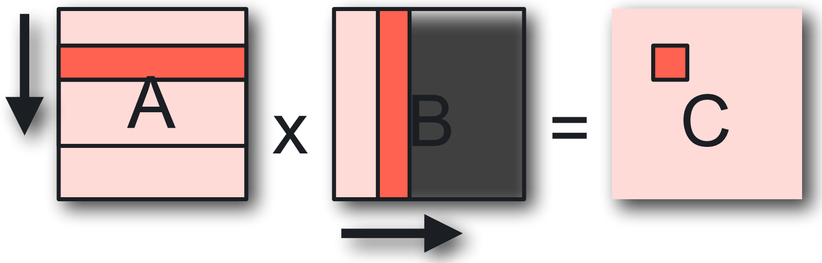
Uses:

- Squeeze computational capacity of the GPU
- Measure efficiency of Cache/SHM optimizations

Source: NVIDIA

Example
Matrix/Matrix Multiplication

Square Matrix Multiplication




```
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    for (int k=0; k<N; k++)
      c[i*N+j] += a[i*N+k] * b[k*N+j];
```

2D Matrices (NxN)

1D (row by column) operation

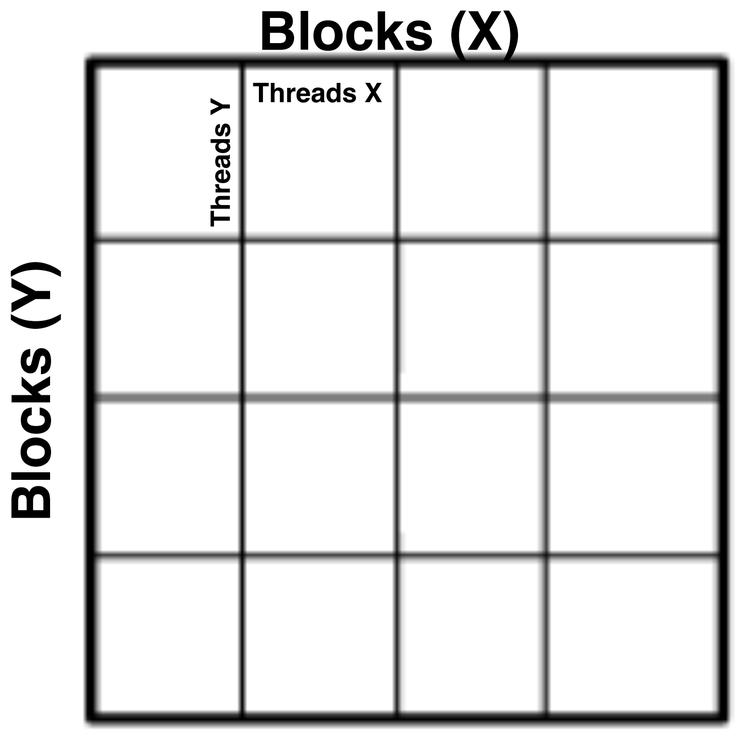
Complexity: O(N^3)

Parallelize 2D Geometry

```
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
```

Blocks (X) Threads (X)

Blocks (Y) Threads (Y)



Each thread Executes a 1D Kernel

```
for (int k=0; k<N; k++)
  c[i*N+j] += a[i*N+k] * b[k*N+j];
```

Naive matrix multiplication kernel

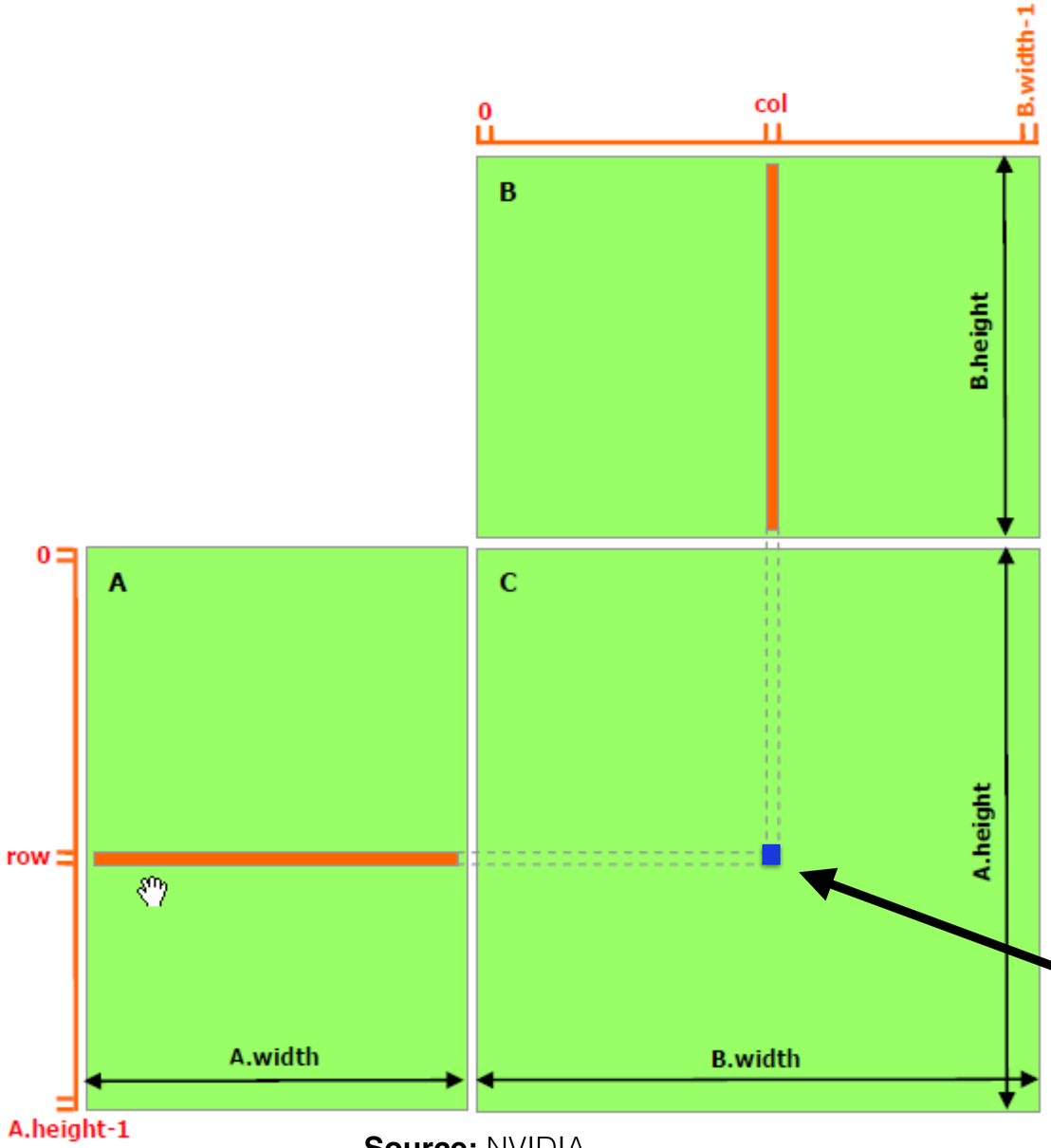
```
 // Host Code
#define BLOCK_SIZE 32
dim3 blocks(N/BLOCK_SIZE, N/BLOCK_SIZE);
dim3 threads(BLOCK_SIZE, BLOCK_SIZE); // 512 Threads per Block
matrix<<< blocks, threads >>>(dev_a, dev_b, dev_c,N);

 // Device Code
 __global__ void matrix(double* A, double* B, double* C, int N)
{
 int x = threadIdx.x + blockIdx.x*blockDim.x;
 int y = threadIdx.y + blockIdx.y*blockDim.y;

 if (x<N && y<N)
 {
  c[x*N + y] = 0;

  for (int k=0; k<N; k++)
   C[x*N + y] += A[x*N + k] * B[k*N + y];
 }
}
```

Memory Access Pattern



Global Memory

Thread Read Pattern

Thread Write Pattern

N updates

Source: NVIDIA

Shared Memory Optimized Kernel

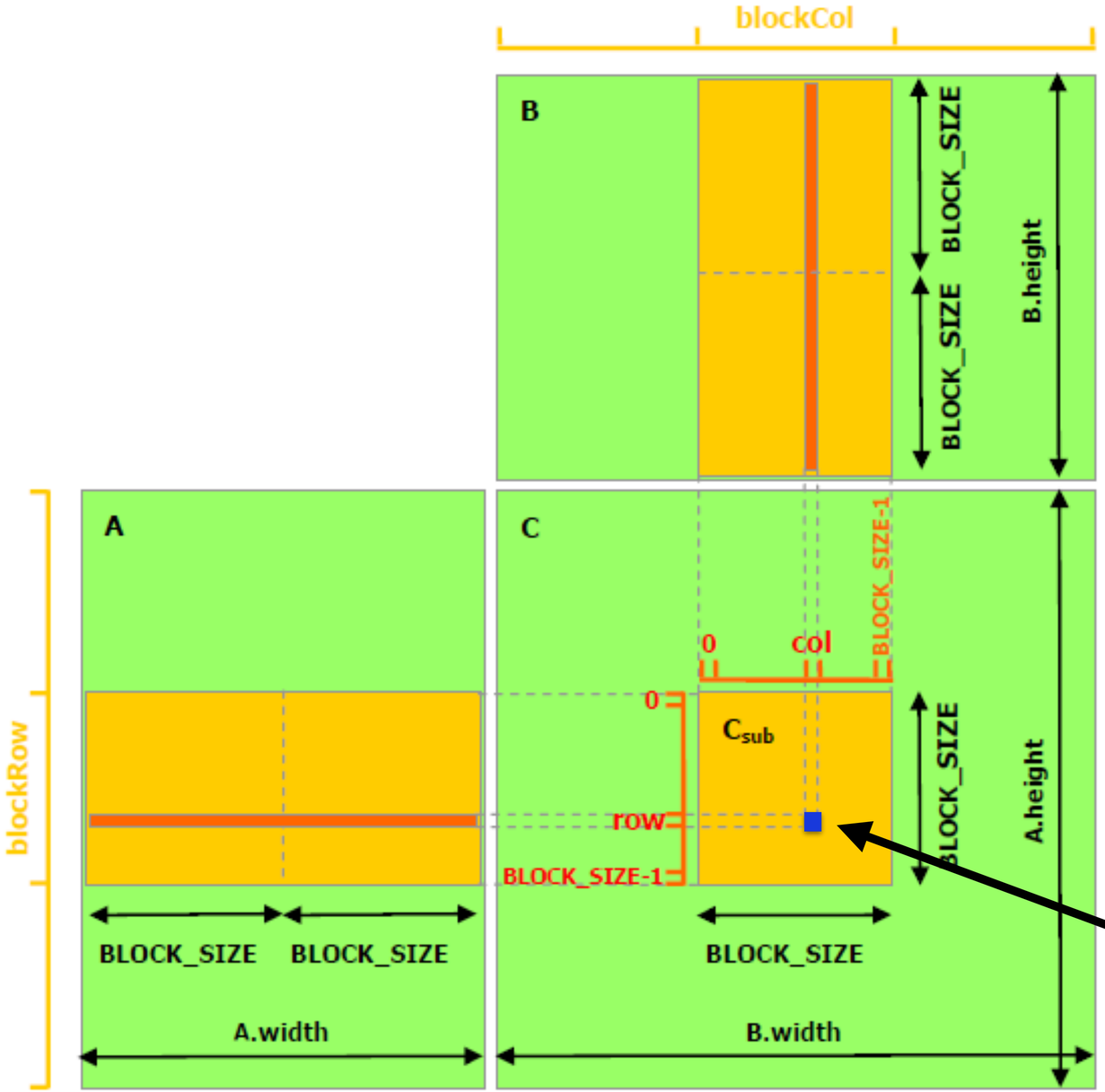
```
__global__ void matrix(double* A, double* B, double* C, int N)
{
 int x = threadIdx.x + blockIdx.x*blockDim.x;
 int y = threadIdx.y + blockIdx.y*blockDim.y;
 int myRow = threadIdx.y
 int myCol = threadIdx.x

 for (int m = 0; m < (N / BLOCK_SIZE); m++)
 {
  __shared__ double As[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ double Bs[BLOCK_SIZE][BLOCK_SIZE];

  As[myRow][myCol] = A[x*N + m*BLOCK_SIZE+myCol];
  Bs[myRow][myCol] = B[(m*BLOCK_SIZE + myRow)*N + y];

  __syncthreads();
  for (int i = 0; i < BLOCK_SIZE; i++) C[x*N + y] += As[myRow][i] * Bs[i][myCol];
  __syncthreads();
 }
}
```

Memory Access Pattern



Global Memory

Shared Memory

Thread Read Pattern

Thread Write Pattern

N updates

Source: NVIDIA

One more optimization:

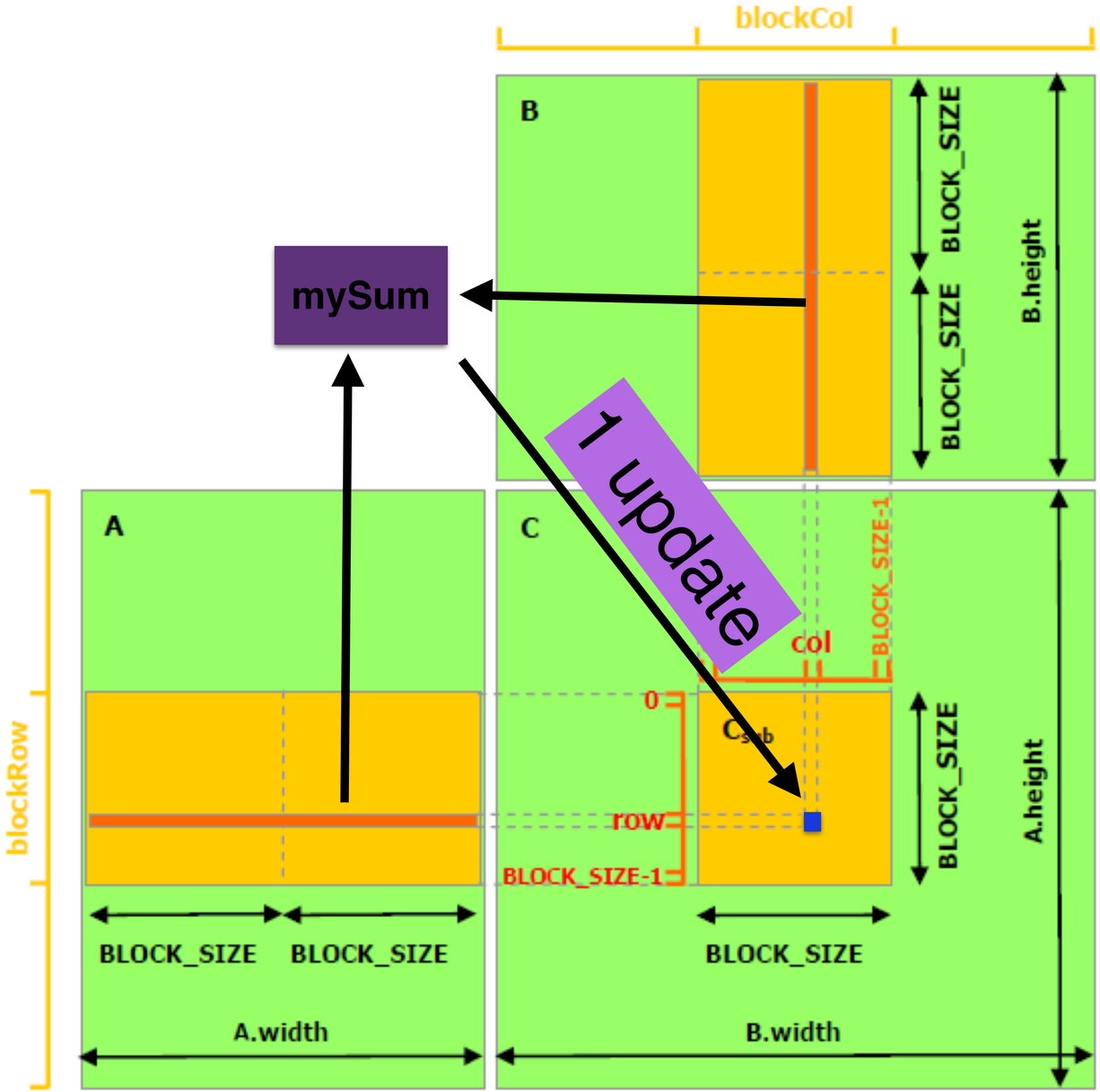
```
__global__ void matrix(double* A, double* B, double* C, int N)
{
 int x = threadIdx.x + blockIdx.x*blockDim.x;
 int y = threadIdx.y + blockIdx.y*blockDim.y;
 int myRow = threadIdx.y
 int myCol = threadIdx.x

 double mySum = 0;
for (int m = 0; m < (N / BLOCK_SIZE); m++)
 {
  __shared__ double As[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ double Bs[BLOCK_SIZE][BLOCK_SIZE];

  As[myRow][myCol] = A[x*N + m*BLOCK_SIZE+myCol];
  Bs[myRow][myCol] = B[(m*BLOCK_SIZE + myRow)*N + y];

  __syncthreads();
  for (int i = 0; i < BLOCK_SIZE; i++) mySum += As[myRow][i] * Bs[i][myCol];
  __syncthreads();
 }
 C[x*N + y] += mySum;
}
```

Memory Access Pattern



Global Memory

Shared Memory

Thread Read Pattern

Register Memory

Thread Write Pattern

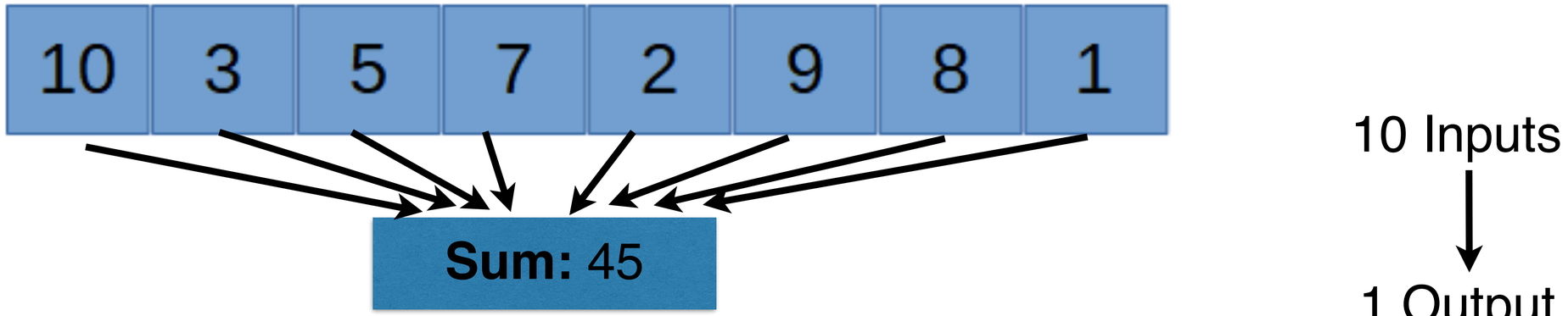
Source: NVIDIA

Kernel Decomposition

Vector Reduction

Fact: Some problems require reading from an entire array but write to a smaller set.

Example: Vector reduction

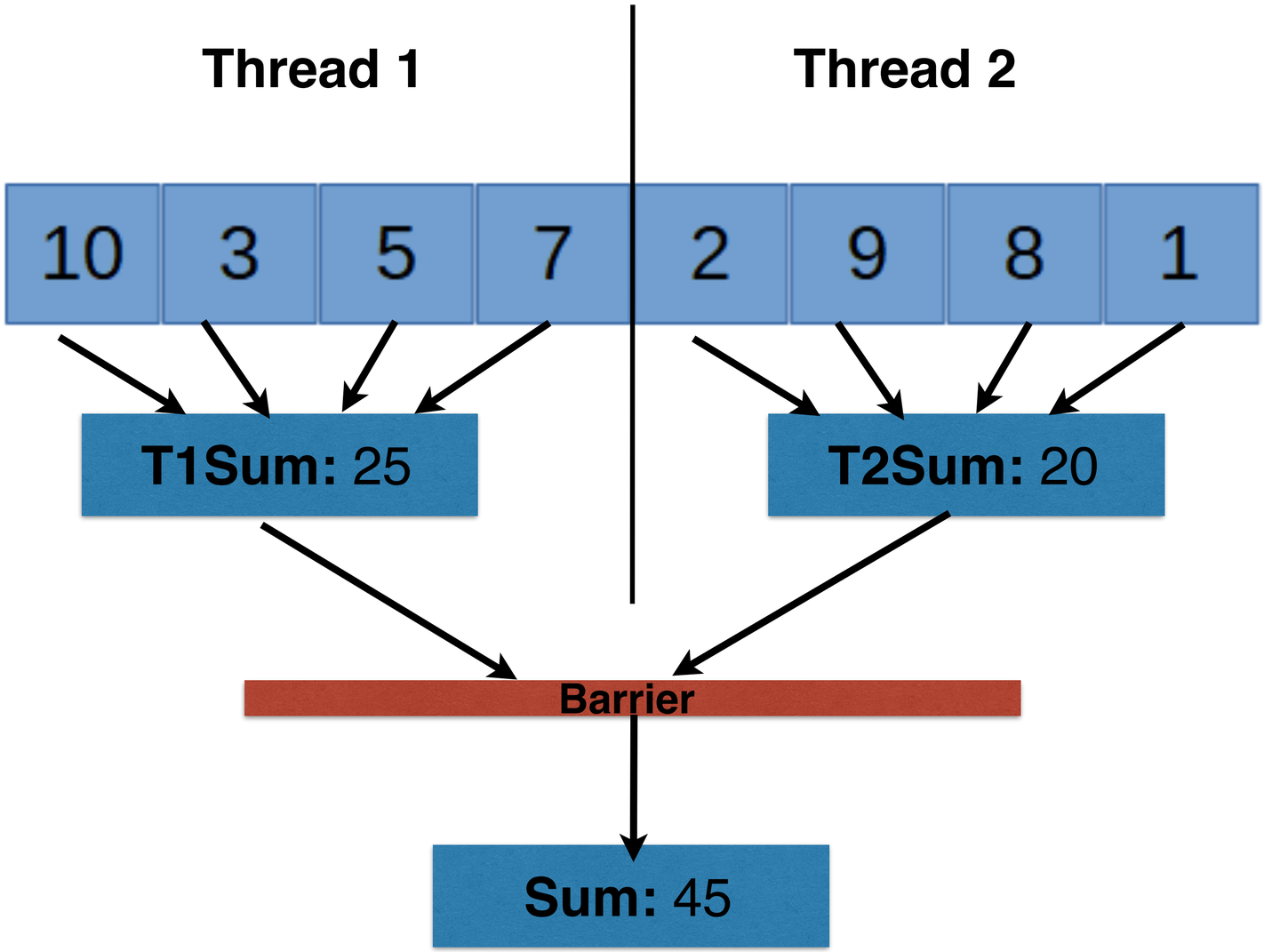


10 Inputs

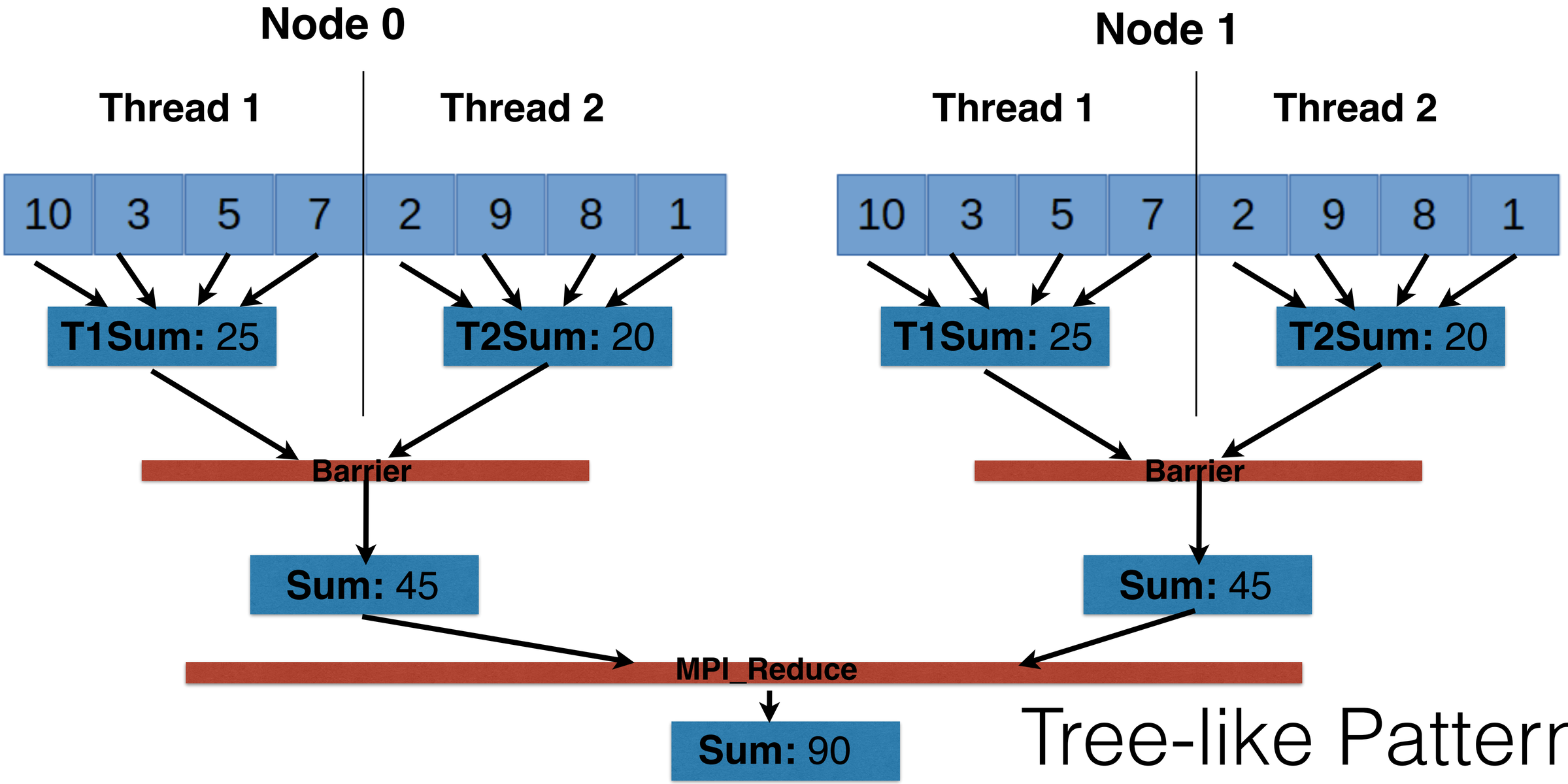
1 Output

```
double mySum = 0.0;
for (int i = 0; i < N; i++) mySum += vector[i];
```

Parallel Vector Reduction



Distributed Parallel Vector Reduction



Tree-like Pattern