

Contemporary Computer Architecture TDSN13

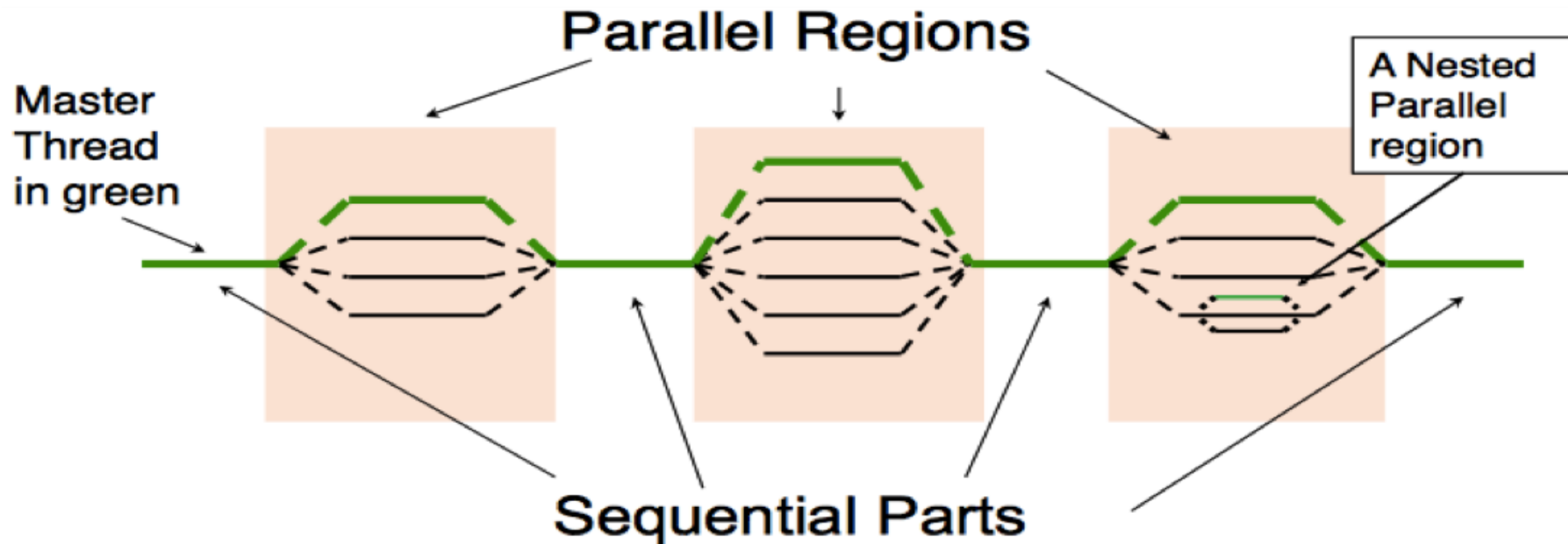
LECTURE 12 – SUMMING UP

ANDREAS AXELSSON (ANDREAS.AXELSSON@JU.SE)

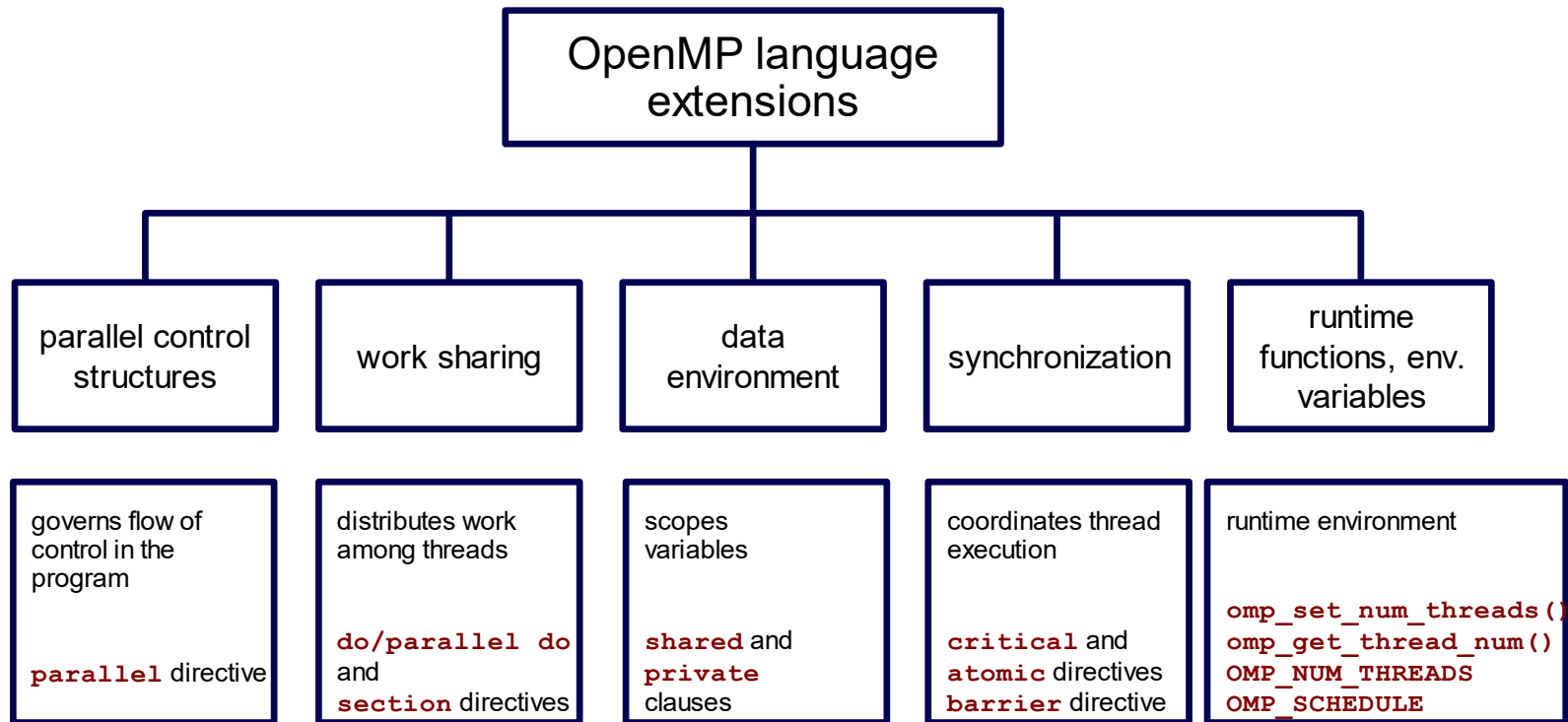
OpenMP

Parallelized code via OpenMP standardized compiler extensions

Support for many architectures and operating systems



OpenMP



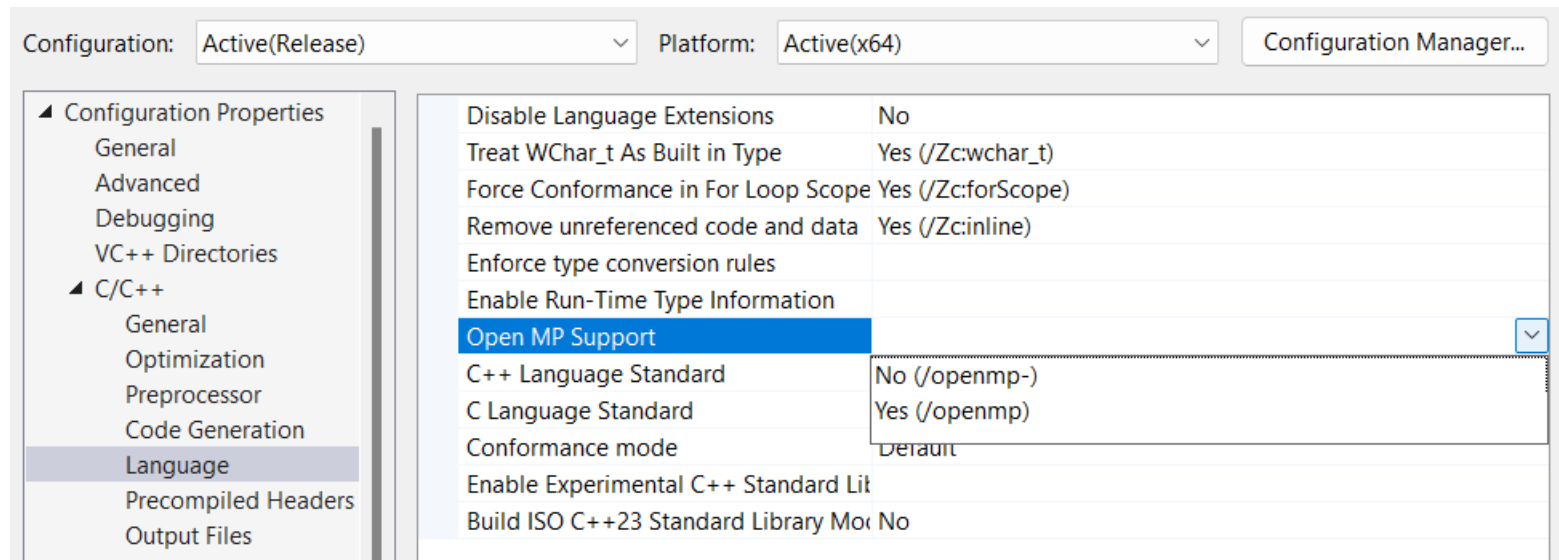
OpenMP

gcc -fopenmp -o HelloOpenMP HelloOpenMP.c

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        int nthreads, thread_id;
        nthreads = omp_get_num_threads();
        thread_id = omp_get_thread_num();
        printf("I have %d thread(s) and my thread id is %d\n", nthreads, thread_id);
    }
}
```

OpenMP in Visual Studio



Message Passing Interface (MPI)

Wikipedia:

- Message Passing Interface (MPI) is a standardized and portable message-passing standard designed to function on parallel computing architectures.
- The MPI standard defines the syntax and semantics of library routines that are useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran.
- There are several open-source MPI implementations, which fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

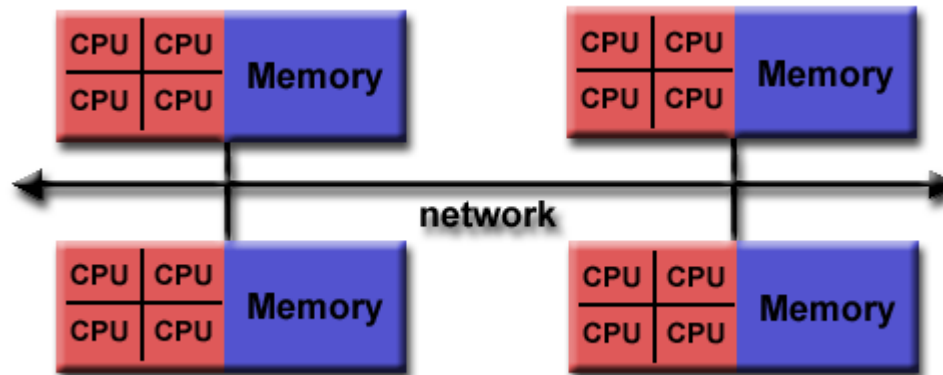
    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processor
s\n",
           processor_name, world_rank, world_size);

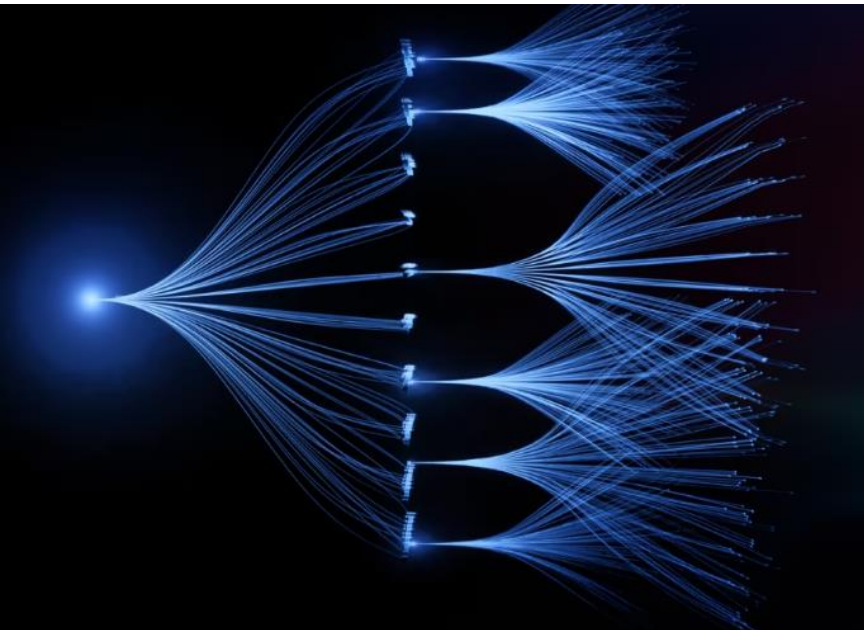
    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Message Passing Interface (MPI)

- **Standardization** - MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability** - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.
- **Functionality** - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
 - NOTE: Most MPI programs can be written using a dozen or less routines
- **Availability** - A variety of implementations are available, both vendor and public domain.



Radeon Open Compute

An abstract graphic on the left side of the slide, featuring a central point from which numerous thin, glowing blue lines radiate outwards, resembling a network or fiber-optic connection. The lines are more densely packed in the center and spread out towards the right, creating a sense of depth and connectivity.

AMD ROCm for AI

AMD ROCm™ software offers a suite of optimizations for AI workloads—from Large Language Models (LLMs), to image / video detection & recognition, life sciences & drug discovery, autonomous driving, robotics, and more—and supports the broader AI software ecosystem including open frameworks, models, and tools.

[Visit AMD ROCm Developer Hub](#)

Heterogeneous Interface for Portability (HIP) \approx CUDA for Radeon


```
R"(
    // This kernel computes FFT of length 1024. The 1024 length FFT is decomposed into
    // calls to a radix 16 function, another radix 16 function and then a radix 4 function

    __kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
                              __local float *sMemx, __local float *sMemy) {
        int tid = get_local_id(0);
        int blockIdx = get_group_id(0) * 1024 + tid;
        float2 data[16];

        // starting index of data to/from global memory
        in = in + blockIdx; out = out + blockIdx;

        globalLoads(data, in, 64); // coalesced global reads
        fftRadix16Pass(data);      // in-place radix-16 pass
        twiddleFactorMul(data, tid, 1024, 0);

        // local shuffle using local memory
        localShuffle(data, sMemx, sMemy, tid, (((tid & 15) * 65) + (tid >> 4)));
        fftRadix16Pass(data);      // in-place radix-16 pass
        twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication

        localShuffle(data, sMemx, sMemy, tid, (((tid >> 4) * 64) + (tid & 15)));

        // four radix-4 function calls
        fftRadix4Pass(data);      // radix-4 function number 1
        fftRadix4Pass(data + 4);  // radix-4 function number 2
        fftRadix4Pass(data + 8);  // radix-4 function number 3
        fftRadix4Pass(data + 12); // radix-4 function number 4

        // coalesced global writes
        globalStores(data, out, 64);
    }
)"
```

OpenCL



```
#include <stdio.h>
#include <time.h>
#include "CL/opencl.h"

#define NUM_ENTRIES 1024

int main() // (int argc, const char* argv[])
{
    // CONSTANTS
    // The source code of the kernel is represented as a string
    // located inside file: "fft1D_1024_kernel_src.cl". For the details see the next listing.
    const char *KernelSource =
        #include "fft1D_1024_kernel_src.cl"
        ;

    // Looking up the available GPUs
    const cl_uint num = 1;
    clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 0, NULL, (cl_uint*)&num);

    cl_device_id devices[1];
    clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, num, devices, NULL);

    // create a compute context with GPU device
    cl_context context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

    // create a command queue
    clGetDeviceIDs(NULL, CL_DEVICE_TYPE_DEFAULT, 1, devices, NULL);
    cl_command_queue queue = clCreateCommandQueue(context, devices[0], 0, NULL);

    // allocate the buffer memory objects
    cl_mem memobjs[] = { clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * 2 *
NUM_ENTRIES, NULL, NULL),
        clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * 2 * NUM_ENTRIES, NULL, NULL) };
}
```

OpenCL



```
// create the compute program
// const char* fft1D_1024_kernel_src[1] = {  };
cl_program program = clCreateProgramWithSource(context, 1, (const char **)& KernelSource, NULL, NULL);

// build the compute program executable
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the compute kernel
cl_kernel kernel = clCreateKernel(program, "fft1D_1024", NULL);

// set the args values

size_t local_work_size[1] = { 256 };

clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
clSetKernelArg(kernel, 2, sizeof(float)*(local_work_size[0] + 1) * 16, NULL);
clSetKernelArg(kernel, 3, sizeof(float)*(local_work_size[0] + 1) * 16, NULL);

// create N-D range object with work-item dimensions and execute kernel
size_t global_work_size[1] = { 256 };

global_work_size[0] = NUM_ENTRIES;
local_work_size[0] = 64; //Nvidia: 192 or 256
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size, local_work_size, 0, NULL, NULL);
}
```

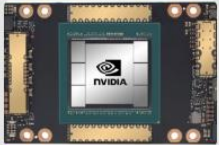



Jetson Nano Developer Kit

GPU	128-core Maxwell
CPU	Quad-core ARM A57 @ 1.43 GHz
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
Storage	microSD (not included)
Video Encode	4K @ 30 4x 1080p @ 30 9x 720p @ 30 (H.264/H.265)
Video Decode	4K @ 60 2x 4K @ 30 8x 1080p @ 30 18x 720p @ 30 (H.264/H.265)
Camera	2x MIPI CSI-2 DPHY lanes
Connectivity	Gigabit Ethernet, M.2 Key E
Display	HDMI and display port
USB	4x USB 3.0, USB 2.0 Micro-B
Others	GPIO, I ² C, I ² S, SPI, UART
Mechanical	69 mm x 45 mm, 260-pin edge connector

472 GFLOPS

NVIDIA A100 80GB PCIe

[NVIDIA GTC May 2020 Keynote Pt6: NVIDIA A100 Data Center GPU Based on NVIDIA Ampere Architecture](#)

	A100	Tesla V100s	Tesla V100	Tesla P100
Picture				
GPU	7nm GA100	12nm GV100	12nm GV100	16nm GP100
Die Size	826 mm ²	815 mm ²	815 mm ²	610 mm ²
Transistors	54 billion	21.1 billion	21.1 billion	15.3 billion
SMs	108	80	80	56
CUDA Cores	6912	5120	5120	3840
Tensor Cores	432	640	640	NA
FP16 Compute	78 TFLOPS	32.8 TFLOPS	31.4 TFLOPS	21.2 TFLOPS
FP32 Compute	19.5 TFLOPS	16.4 TFLOPS	15.7 TFLOPS	10.6 TFLOPS
FP64 Compute	9.7 TFLOPS	8.2 TFLOPS	7.8 TFLOPS	5.3 TFLOPS
Boost Clock	~1410MHz	~1601 MHz	~1533 MHz	~1480MHz
Max. Memory Bandwidth	1134 GB/s	1134 GB/s	900 GB/s	721 GB/s
Eff. Memory Clock	2430 MHz	2214 MHz	1760 MHz	1408 MHz
Memory Config.	40GB HBM2e	32GB HBM2	16GB / 32GB HB...	16GB HBM2
Memory Bus	5120-bit	4096-bit	4096-bit	4096-bit
TDP	400	250W	300W	300W
Form Factor	SXM4 / PCIe 4.0	PCIe 3.0	SXM2 / PCIe 3.0	SXM

	A100 80GB PCIe
FP64	9.7 TFLOPS
FP64 Tensor Core	19.5 TFLOPS
FP32	19.5 TFLOPS
Tensor Float 32 (TF32)	156 TFLOPS 312 TFLOPS*
BFLOAT16 Tensor Core	312 TFLOPS 624 TFLOPS*
FP16 Tensor Core	312 TFLOPS 624 TFLOPS*
INT8 Tensor Core	624 TOPS 1248 TOPS*
GPU Memory	80GB HBM2e
GPU Memory Bandwidth	1,935 GB/s
Max Thermal Design Power (TDP)	300W
Multi-Instance GPU	Up to 7 MIGs @ 10GB

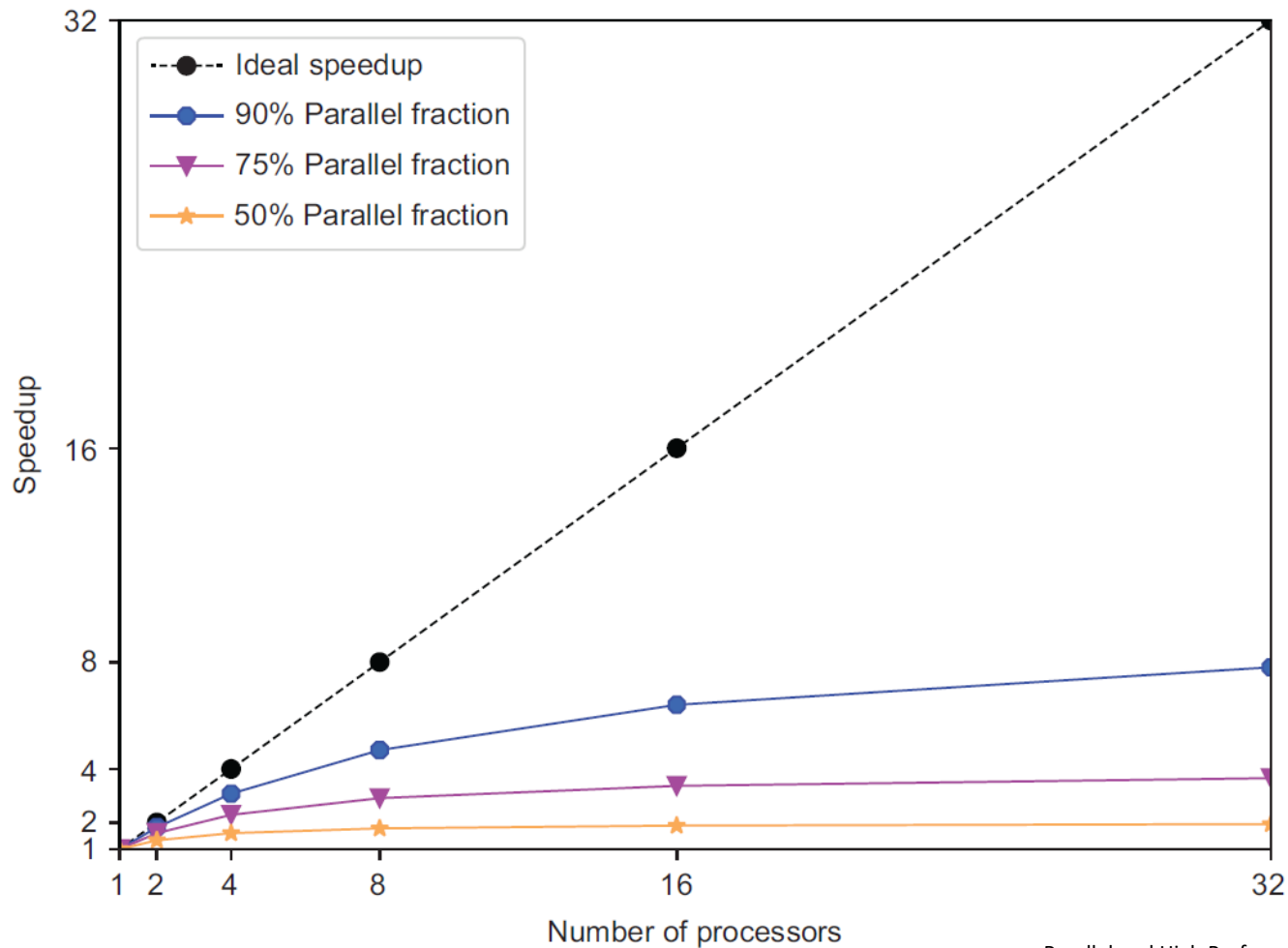
<https://www.nvidia.com/en-us/data-center/a100>

Flynn's Taxonomy

		Instruction	
		Single	Multiple
Data	Single	SISD Single instruction single data	MISD Multiple instruction single data
	Multiple	SIMD Single instruction multiple data	MIMD Multiple instruction multiple data

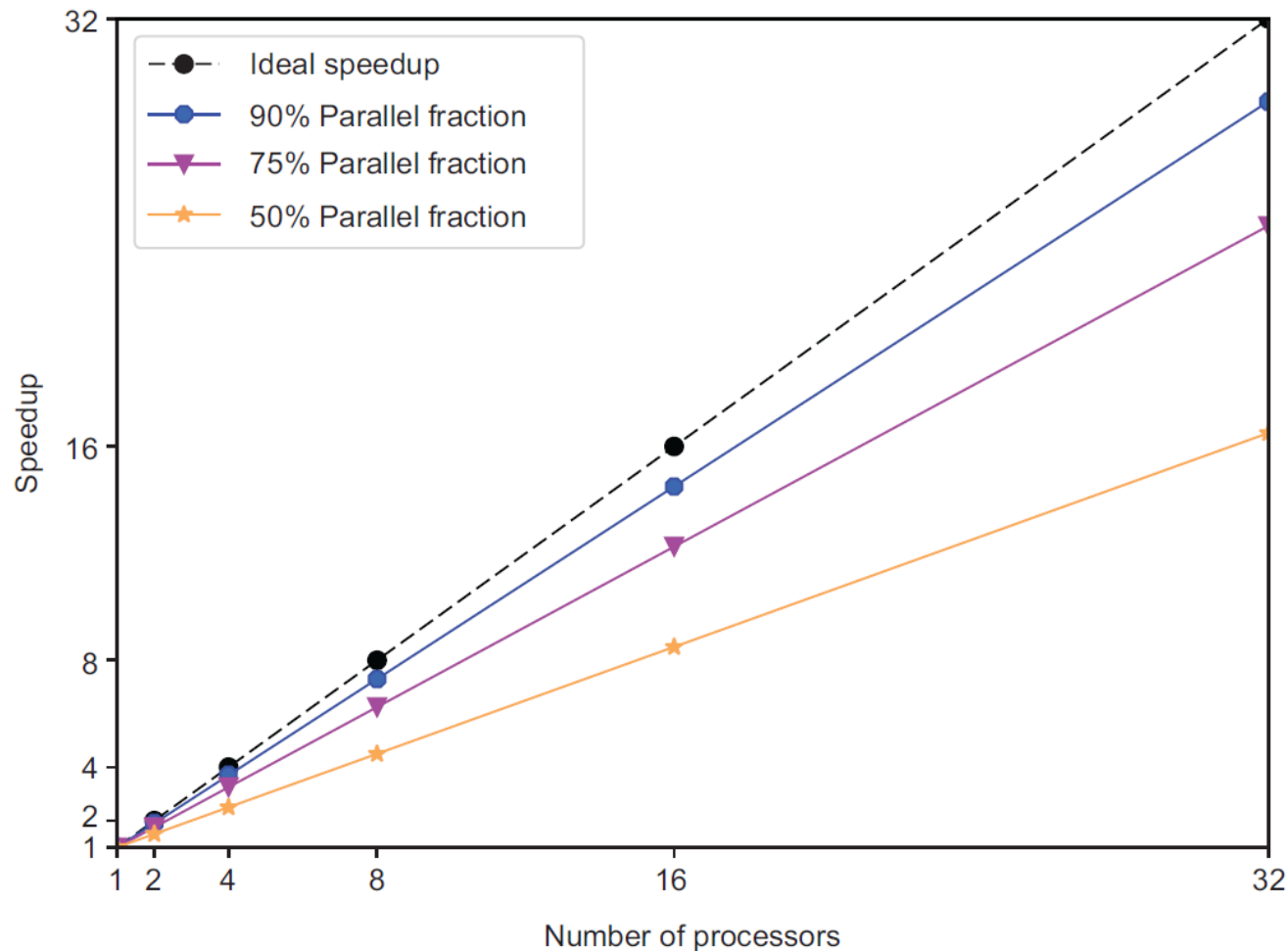
Amdahl's Law

$$SpeedUp(N) = \frac{1}{S + \frac{P}{N}} \text{ where } S+P=1$$



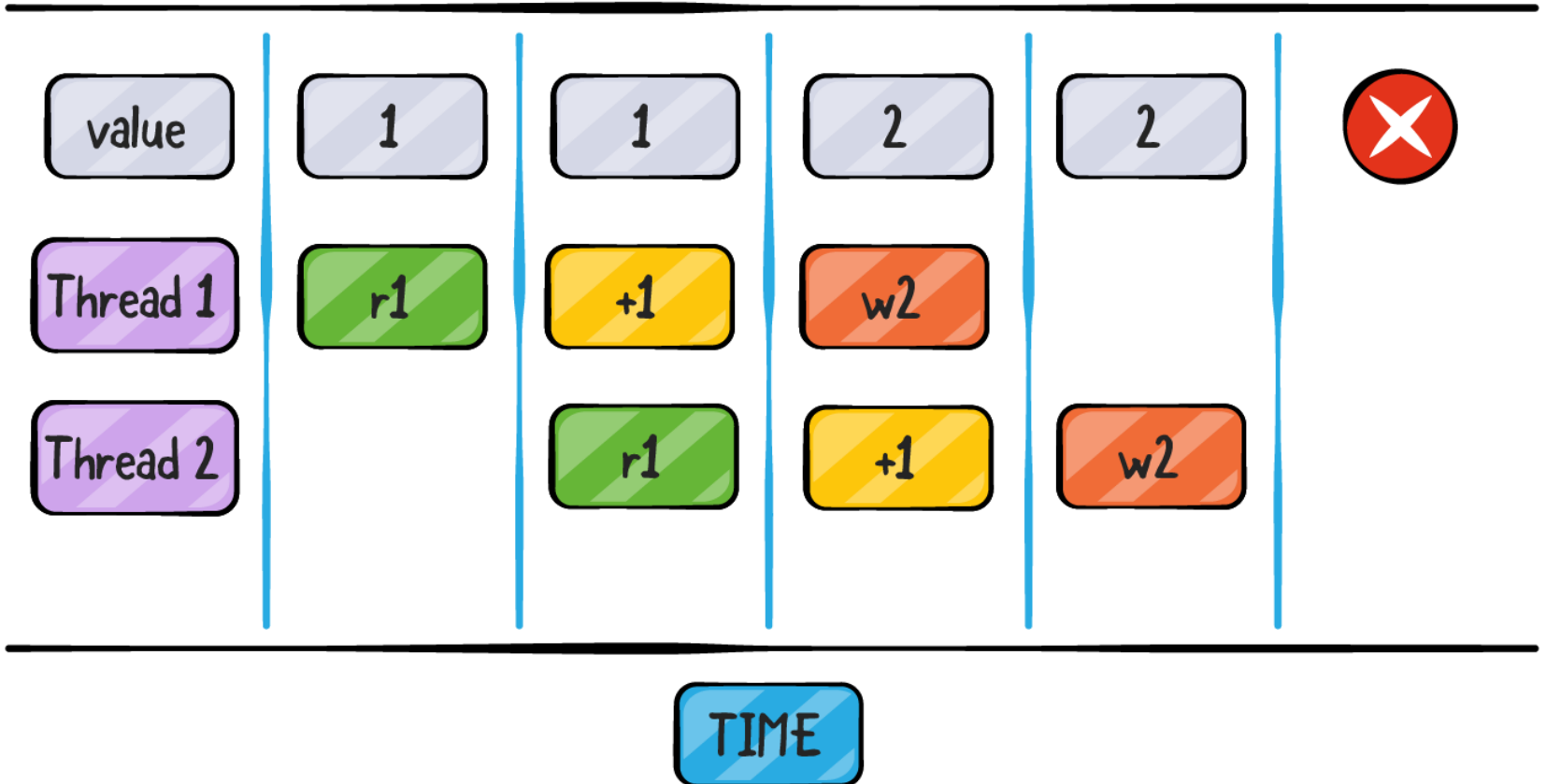
Gustafson-Barsis's Law

$SpeedUp(N) = N - S * (N - 1)$ where S is a fraction

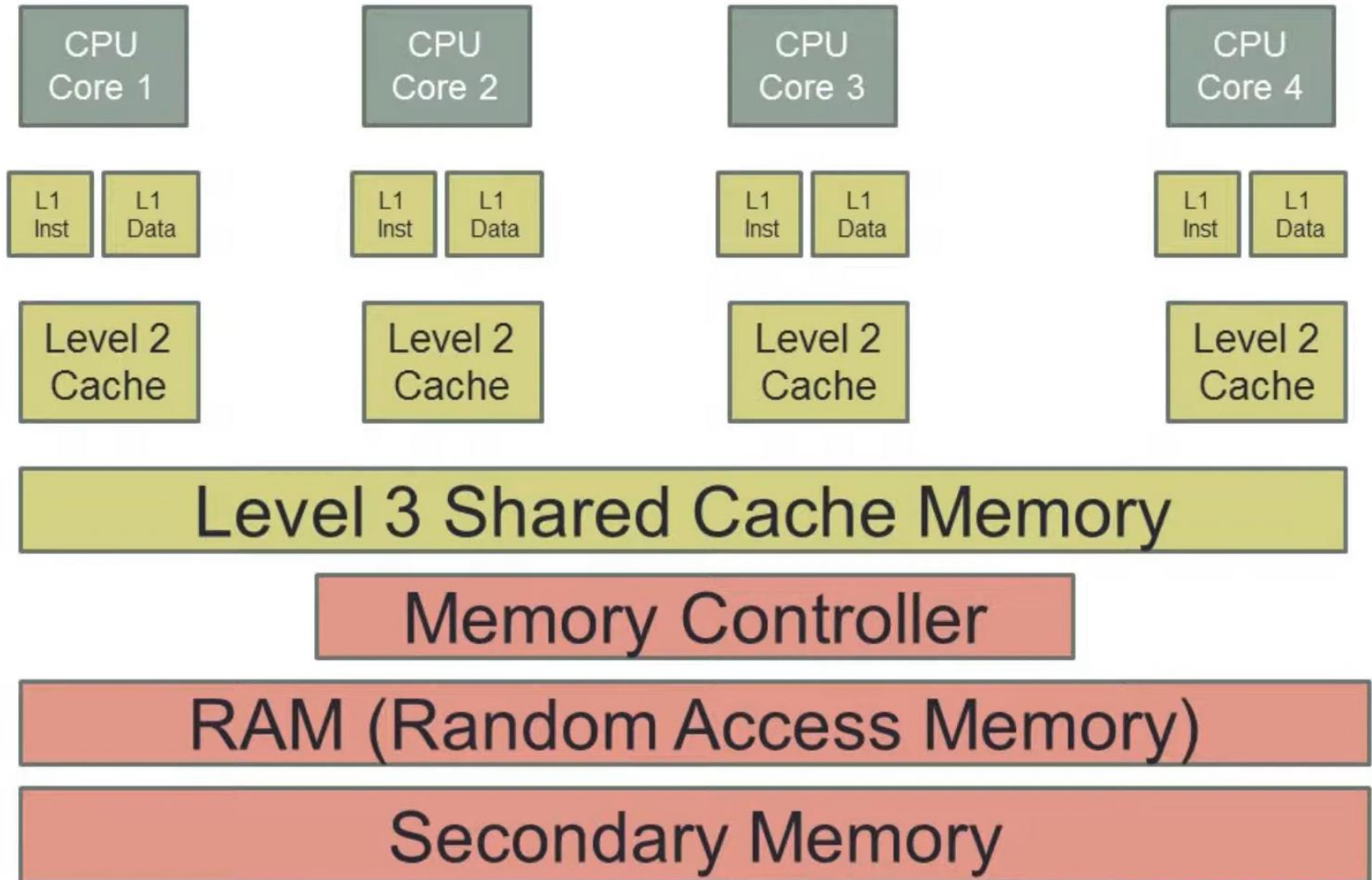


Race Conditions

RACE CONDITION

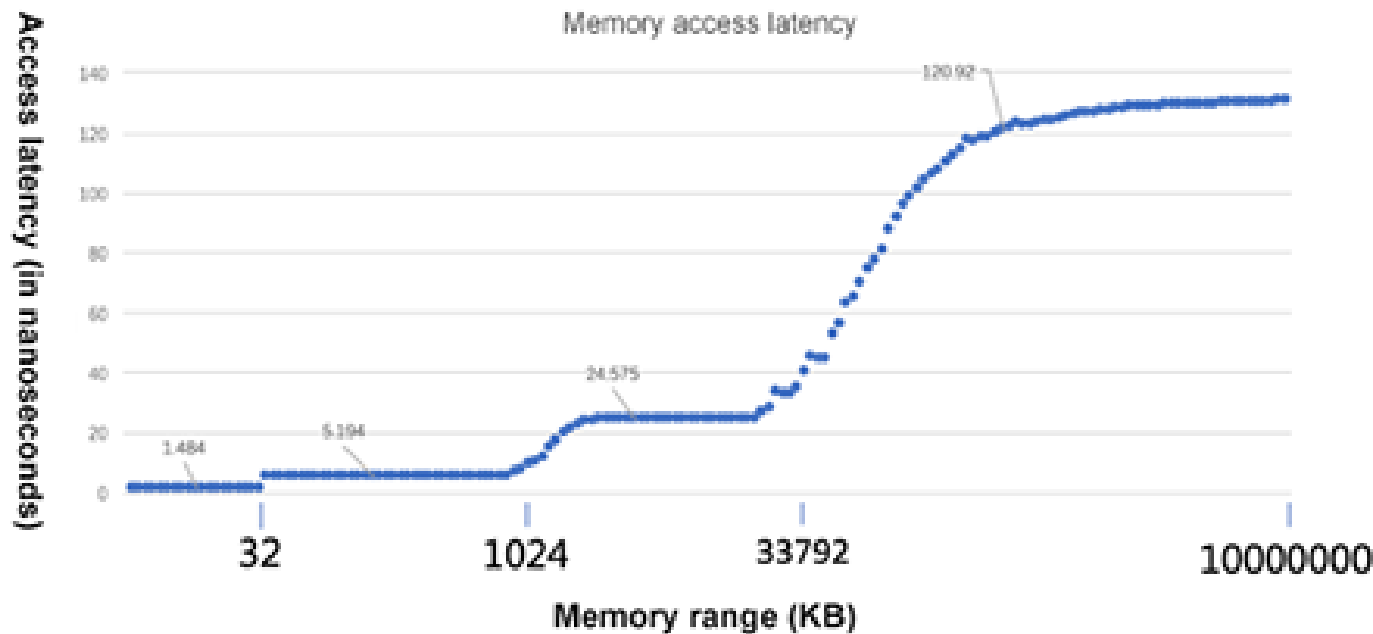


Cache memory



Cache performance differences

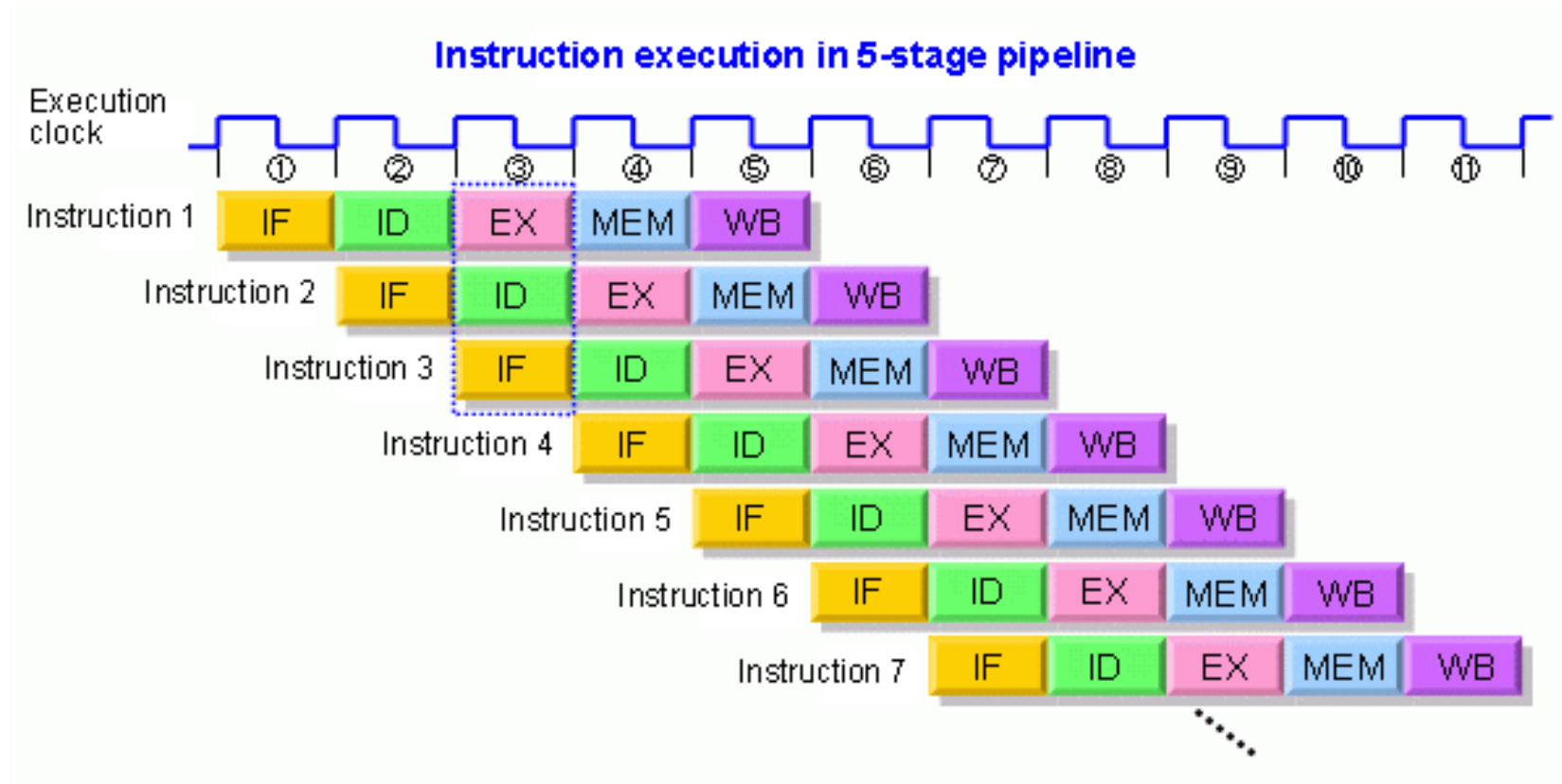
Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz



Test tool: LMBench

Test method: lat_mem_rd 8000 512

5 Stage Pipelining



Introduction to CUDA

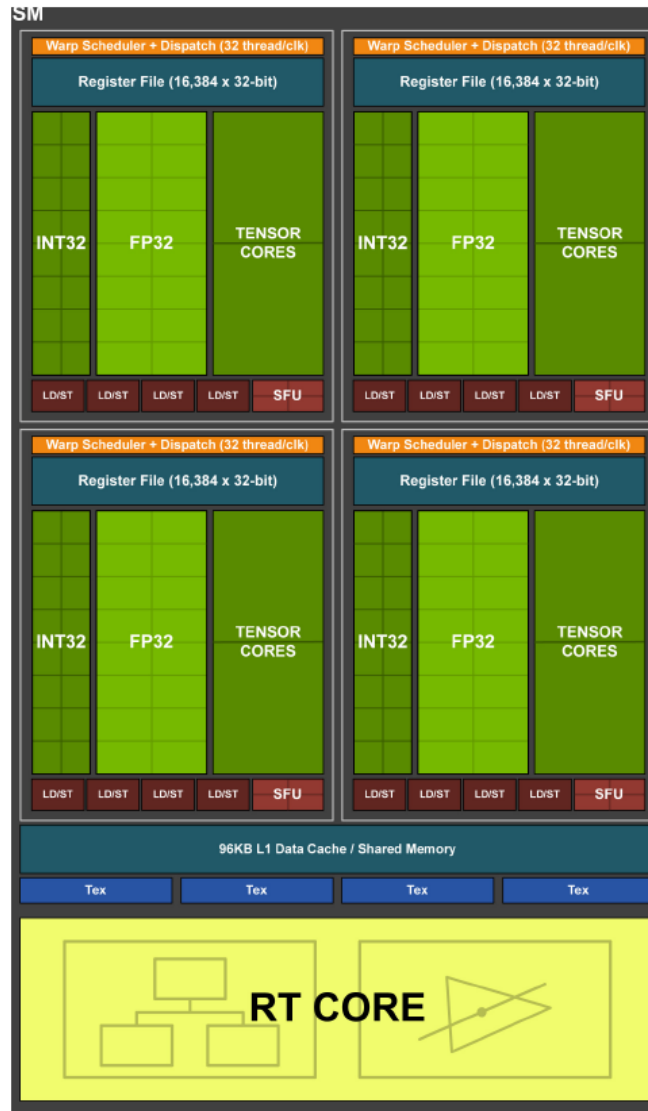
- CUDA Architecture
 - Expose GPU parallelism for general-purpose computing
 - Retain performance
- CUDA C/C++
 - Based on industry-standard C/C++
 - Small set of extensions to enable heterogeneous programming
 - Straightforward APIs to manage devices, memory etc.
- How to get started

Nvidia – GPU Architectures



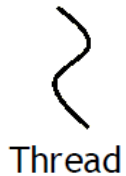
- Tesla
- Fermi
- Kepler
- Maxwell
- Pascal
- Volta
- Turing
- Ampere?

Turing Streaming Multiprocessor



Execution Model

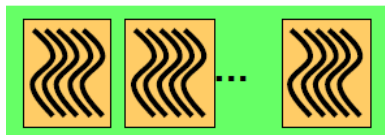
Software



Thread

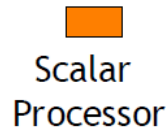


Thread Block

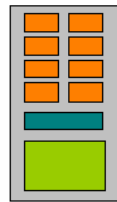


Grid

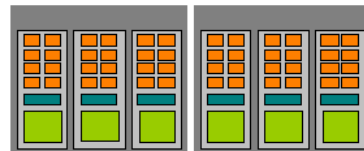
Hardware



Scalar Processor



Multiprocessor



Device

Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

Memory Coalescing

Global memory access happens in transactions of 32 or 128 bytes

The hardware will try to reduce to as few transactions as possible

Coalesced access:

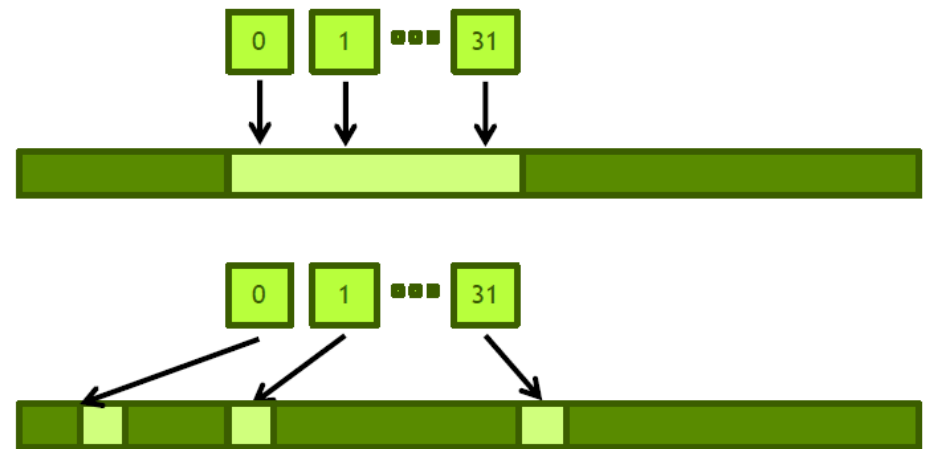
A group of 32 contiguous threads (“warp”) accessing adjacent words

Few transactions and high utilization

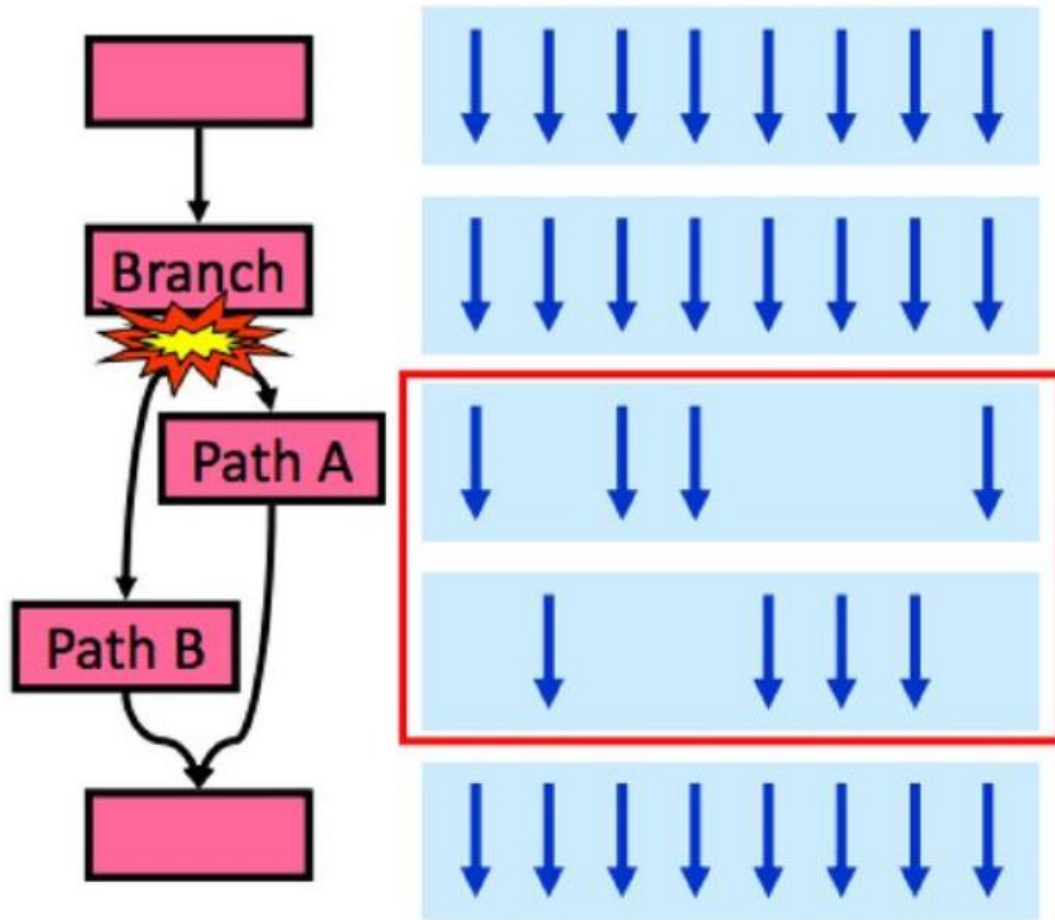
Uncoalesced access:

A warp of 32 threads accessing scattered words

Many transactions and low utilization

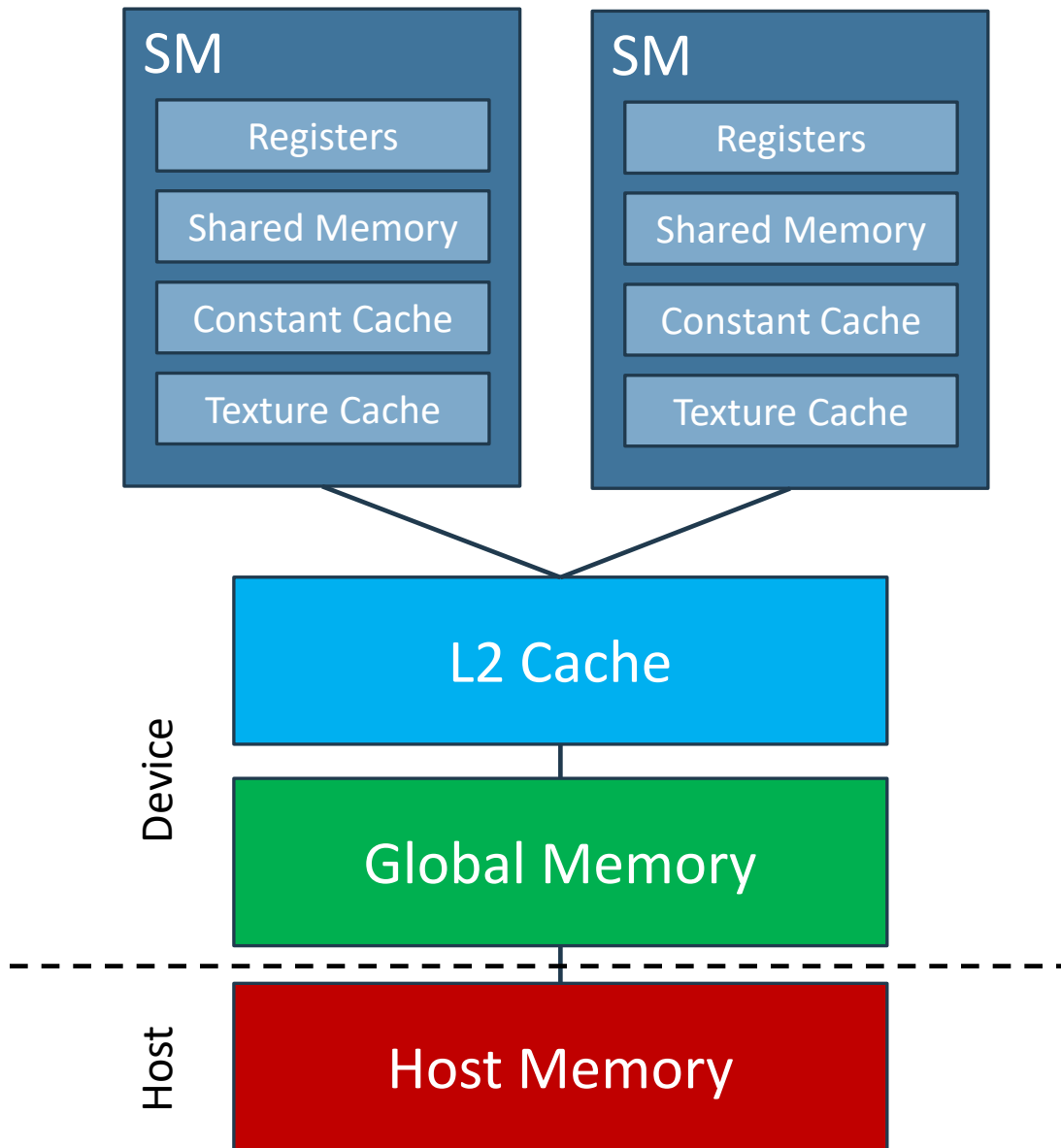


Warp Divergence



50% performance hit!

Memory Hierarchy

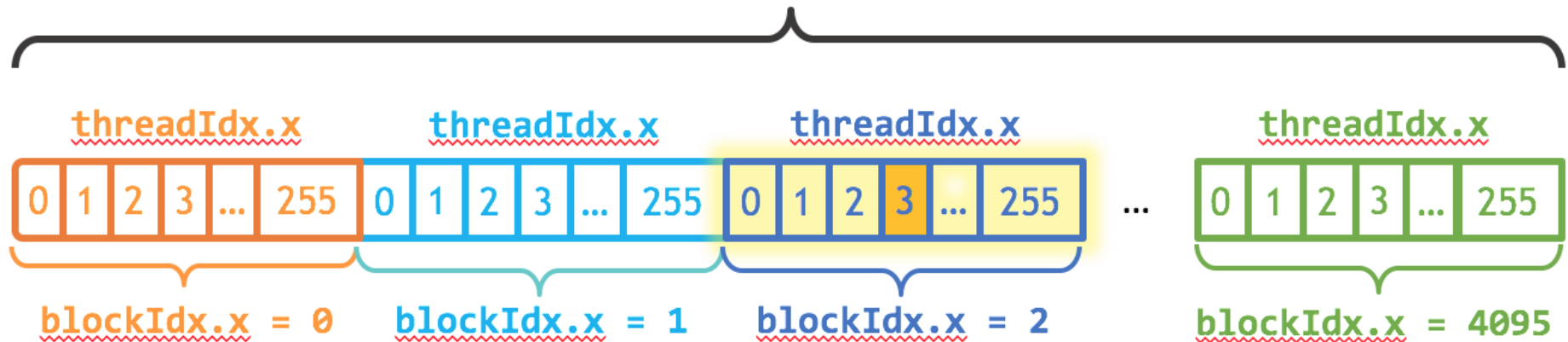


- Global Memory: Large and high latency
- L2 Cache: Medium latency
- SM Caches: Lower latency
- Registers: Lowest latency

Indexing

```
int blockSize = 256;  
int numBlocks = (N + blockSize - 1) / blockSize;  
add<<<numBlocks, blockSize>>>(N, x, y);
```

gridDim.x = 4096

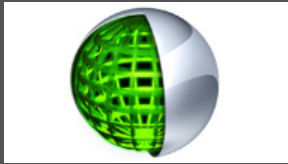


$\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$

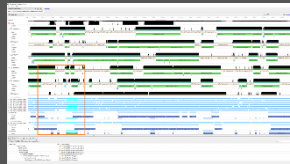
index = (2) * (256) + (3) = 515

Developer Tools – Debuggers

Nsight



Nsight
Systems



CUDA-GDB



CUDA
MEMCHECK



NVIDIA Provided

arm
FORGE

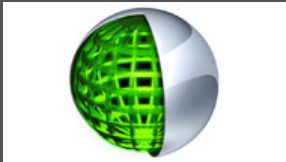
TotalView®

3rd Party

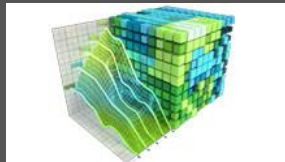
<https://developer.nvidia.com/debugging-solutions>

Developer Tools – Profilers

NSIGHT



NVVP

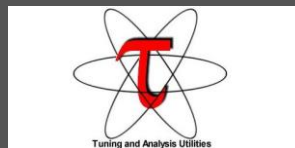


NVPROF

```
==20561== Profiling result:
Time(s)    Time    Calls    Avg      Min      Max  Name
49.88s  866.69ms  584758  1.7170us  1.5040us  2.0160us void th
int_thrust::detail::device_generate_func<thrust::detail::fill_
25.32s  440.40ms  252662  1.7440us  1.5300us  2.3600us void th
t_thrust::detail::device_generate_func<thrust::detail::fill_fu
17.07s  296.60ms  200    1.4830ms  1.2840ms  1.7233ms kerComp
2.90s   21.43ms  200    259.09us  246.97us  264.83us kerMake
1.16s   20.173ms  503    40.265us  928ms  17.677ms [CUDA w
0.93s   16.190ms  200    80.991us  71.840us  96.751us kerColl
0.72s   12.436ms  400    31.509us  14.720us  56.43us [CUDA w
0.69s   12.075ms  200    60.370us  59.600us  62.384us kerMemu
0.63s   10.993ms  200    54.963us  52.600us  58.200us kerMake
0.32s   5.554ms  200    27.761us  22.550us  33.155us [CUDA w
0.12s   2.1342ms  1      2.1342ms  2.1342ms  2.1342ms void th
```

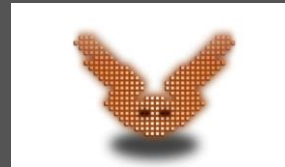
NVIDIA Provided

TAU



Tuning and Analysis Utilities

VampirTrace

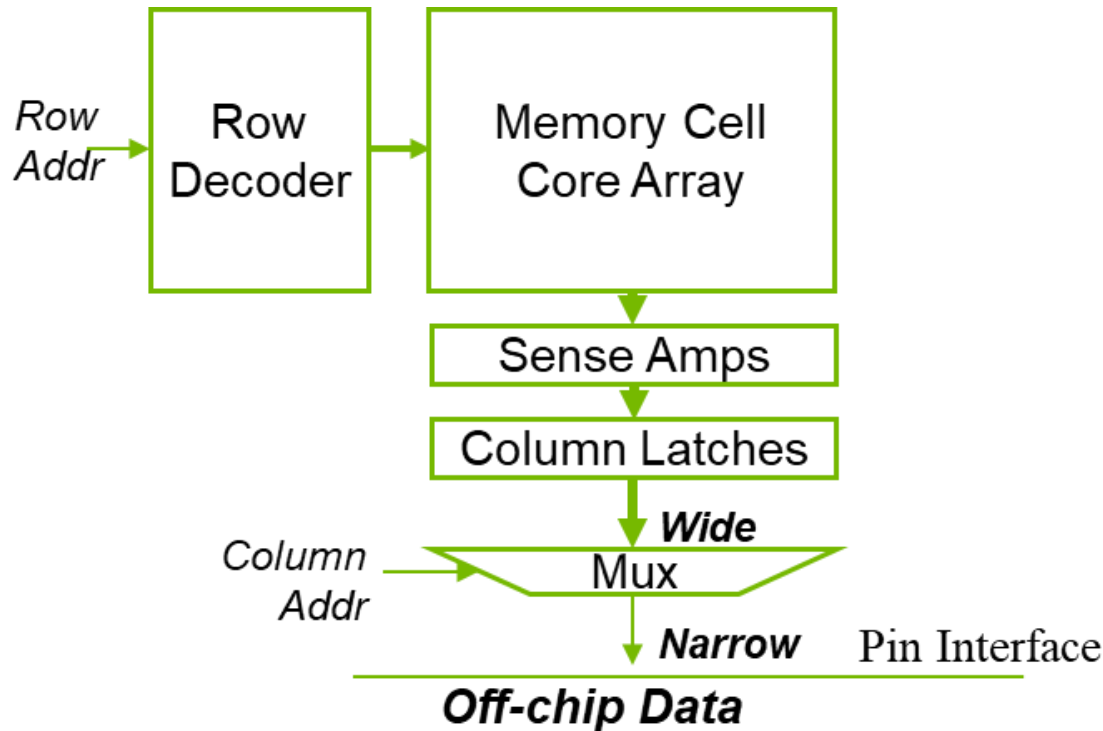


3rd Party

<https://developer.nvidia.com/performance-analysis-tools>

DRAM Core Array Organization

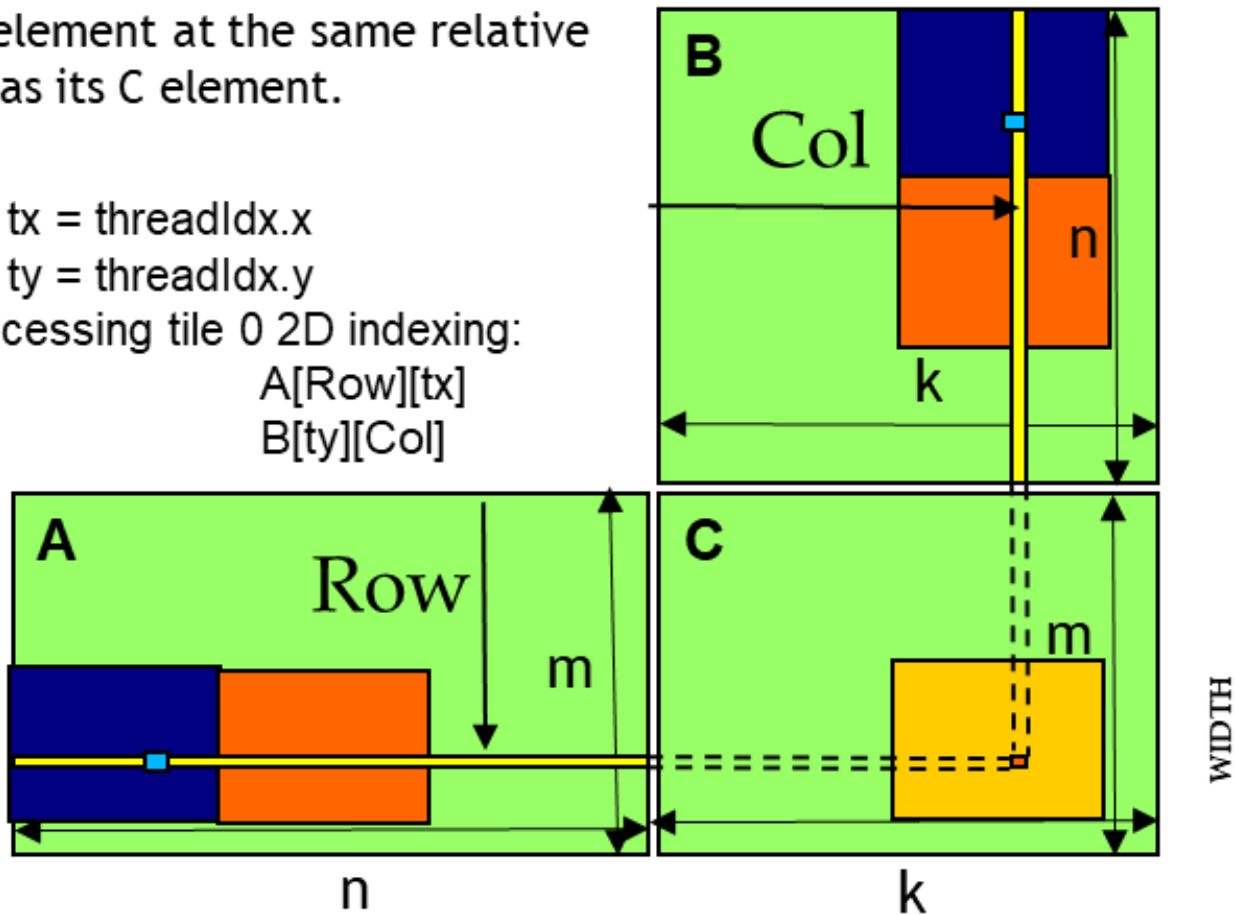
- Each DRAM core array has about 16M bits
- Each bit is stored in a tiny capacitor made of one transistor



Corner Turning – Loading a tile

Have each thread load an A element and a B element at the same relative position as its C element.

```
int tx = threadIdx.x
int ty = threadIdx.y
Accessing tile 0 2D indexing:
    A[Row][tx]
    B[ty][Col]
```



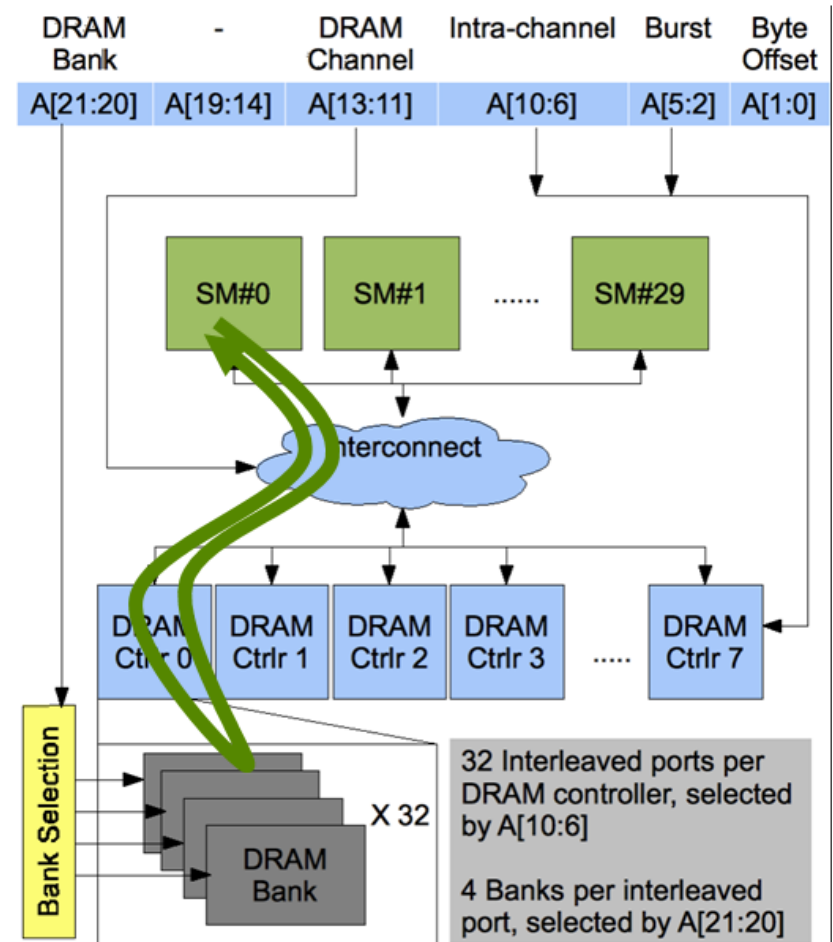
Atomic Operations in CUDA

- Performed by calling functions that are translated into single instructions (a.k.a. *intrinsic functions* or *intrinsics*)
 - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
 - Read CUDA C programming Guide for details
- Atomic Add
 - ```
int atomicAdd(int* address, int val);
```
  - reads the 32-bit word **old** from the location pointed to by **address** in global or shared memory, computes (**old + val**), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

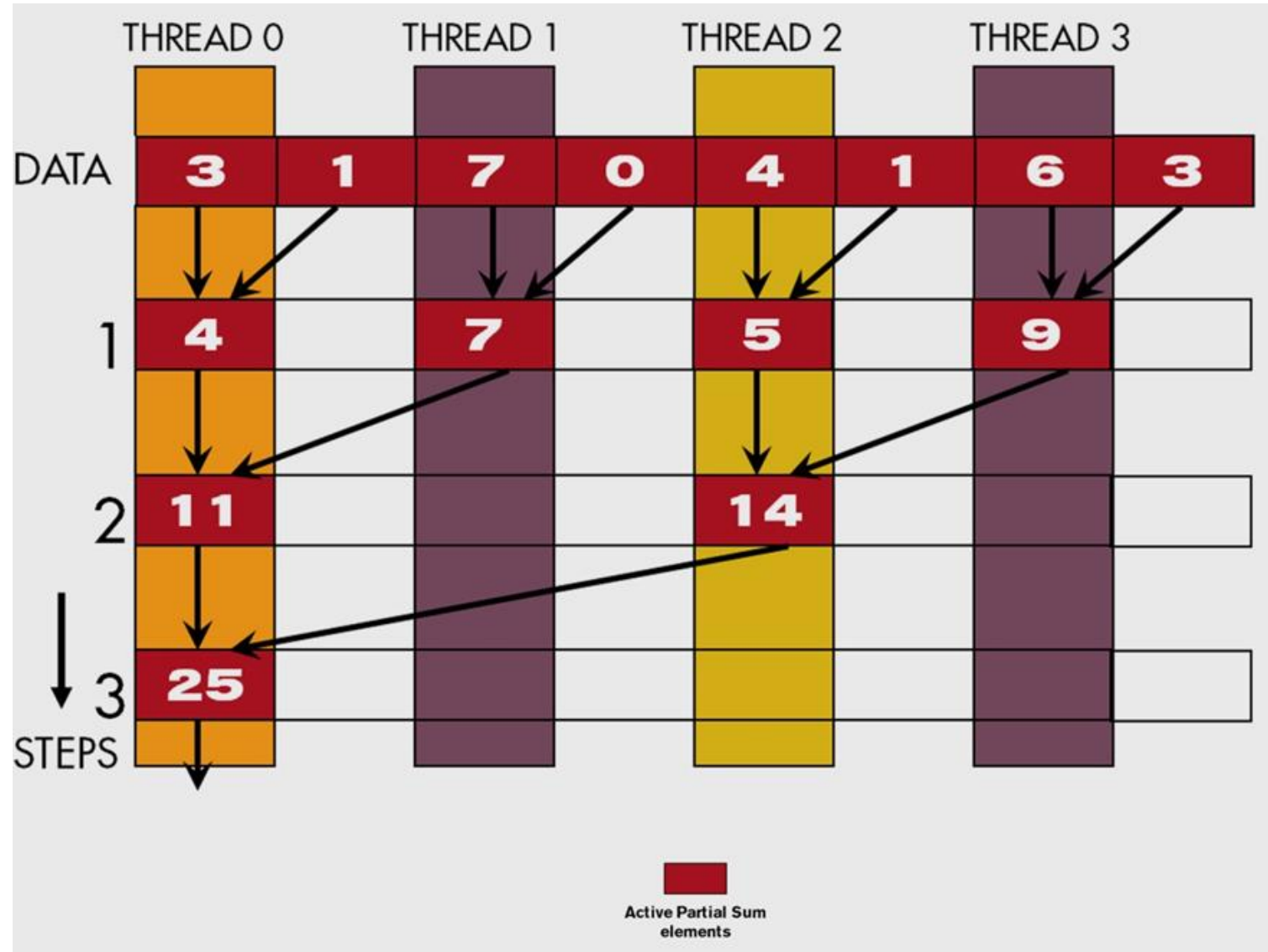
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions>

# Atomic Operation on Global Mem

- An atomic operation on a DRAM location starts with a read, which has a latency of a few hundred cycles
- The atomic operation ends with a write to the same location, with a latency of a few hundred cycles
- During this whole time, no one else can access the location



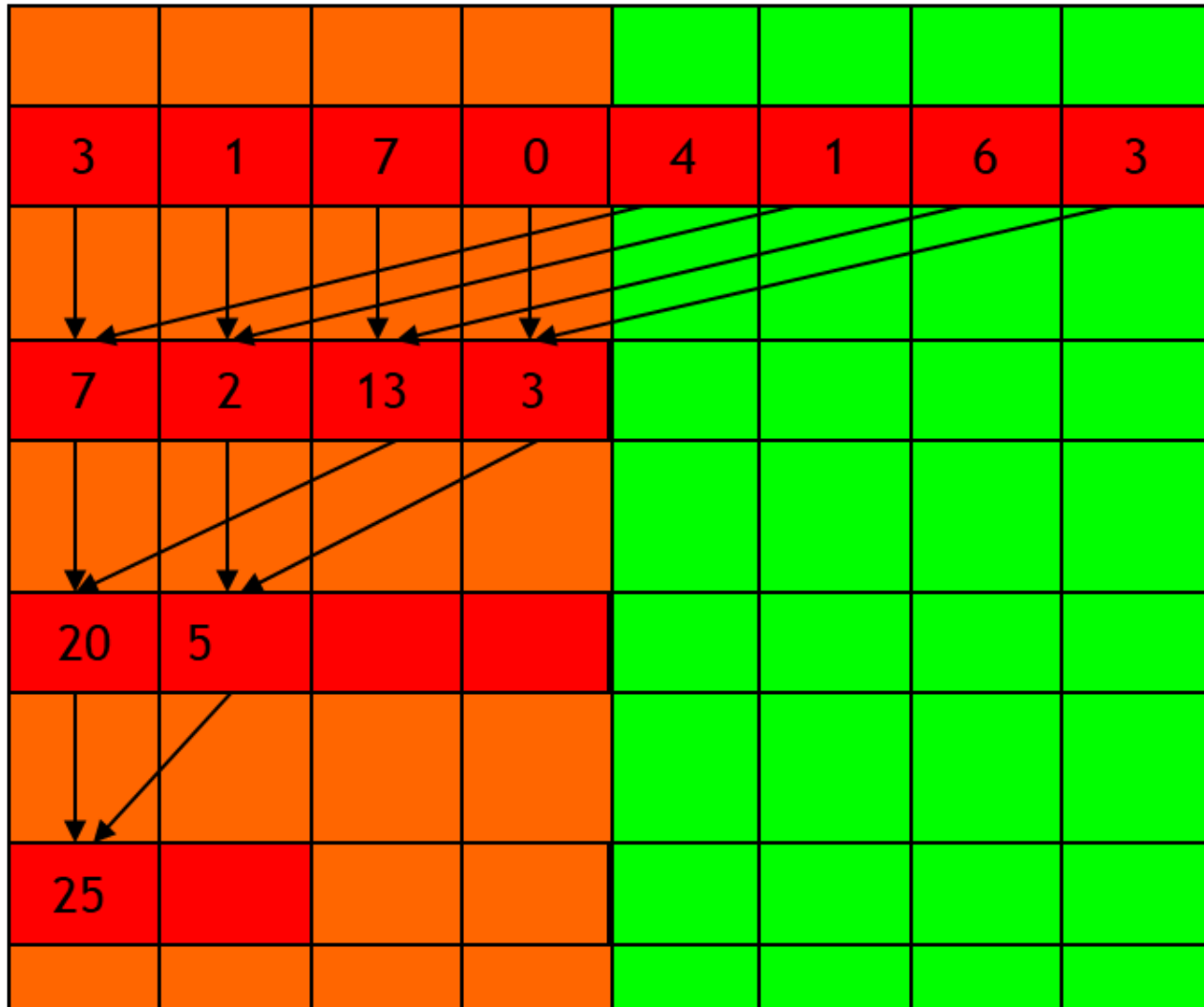
# A Parallel Sum Reduction Example



# An Example of 4 threads

---

Thread 0 Thread 1 Thread 2 Thread 3



# Cuda Libraries

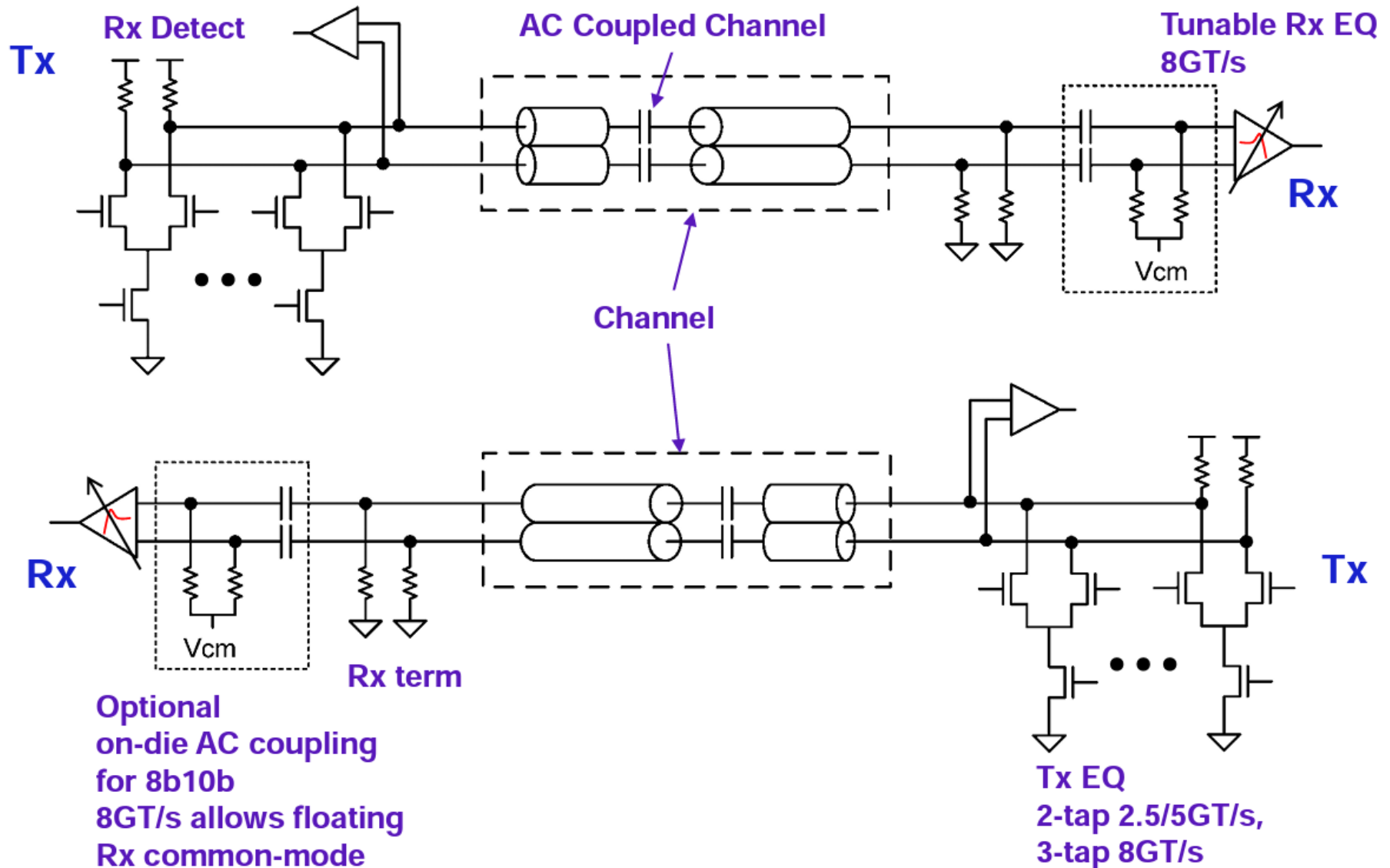
---



Some standard libraries for scientific calculations:

- cuBLAS
- cuSolver
- cuFFT
- Thrust

# High speed serial busses (PCIe etc)



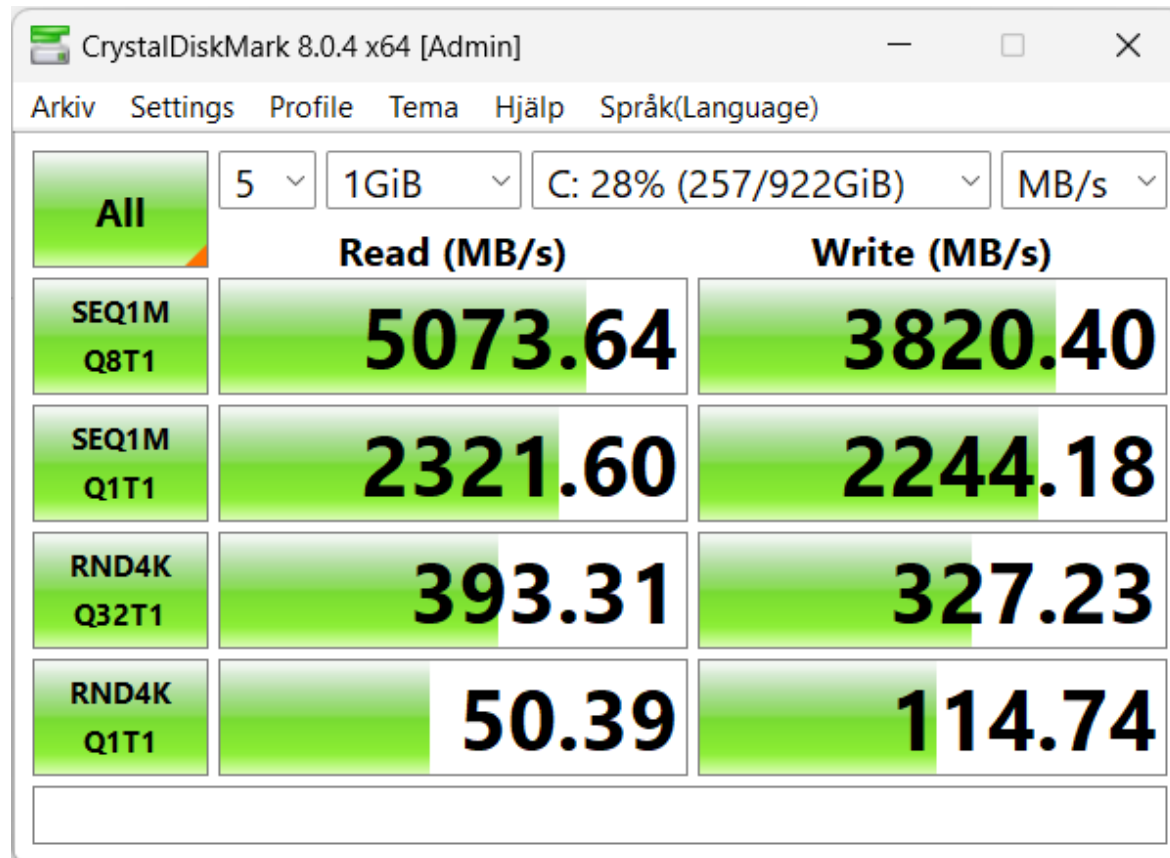
# High speed serial busses (PCIe etc)

---

| PCI Express: Unidirectional Bandwidth in x1 and x16 Configurations |                 |                    |              |               |
|--------------------------------------------------------------------|-----------------|--------------------|--------------|---------------|
| Generation                                                         | Year of Release | Data Transfer Rate | Bandwidth x1 | Bandwidth x16 |
| PCIe 1.0                                                           | 2003            | 2.5 GT/s           | 250 MB/s     | 4.0 GB/s      |
| PCIe 2.0                                                           | 2007            | 5.0 GT/s           | 500 MB/s     | 8.0 GB/s      |
| PCIe 3.0                                                           | 2010            | 8.0 GT/s           | 1 GB/s       | 16 GB/s       |
| PCIe 4.0                                                           | 2017            | 16 GT/s            | 2 GB/s       | 32 GB/s       |
| PCIe 5.0                                                           | 2019            | 32 GT/s            | 4 GB/s       | 64 GB/s       |
| PCIe 6.0                                                           | 2021            | 64 GT/s            | 8 GB/s       | 128 GB/s      |

**Table:** PCI-SIG introduced the first generation of PCI Express in 2003. With each new generation comes a doubling of data transfer rate and total bandwidth per lane configuration, the latter of which is expressed in both unidirectional and bidirectional measurements, depending on the source. To find the total unidirectional bandwidth for each lane configuration, simply multiply the x1 bandwidths listed in the table above by two, four, eight or 16. Multiply the number resulting from that calculation by two to calculate total bidirectional bandwidth. *Source: PCI-SIG*

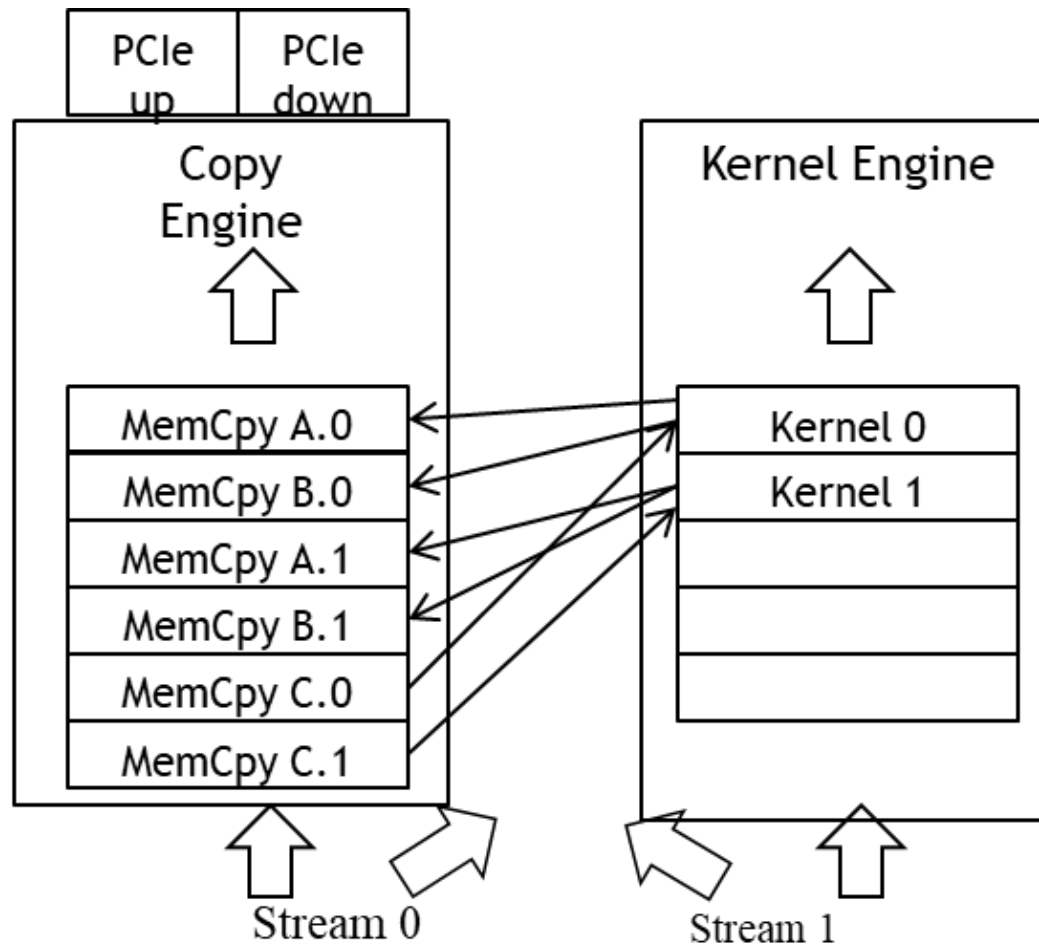
# NVMe disk interface





# C.0 no longer blocks A.1 and B.1

---

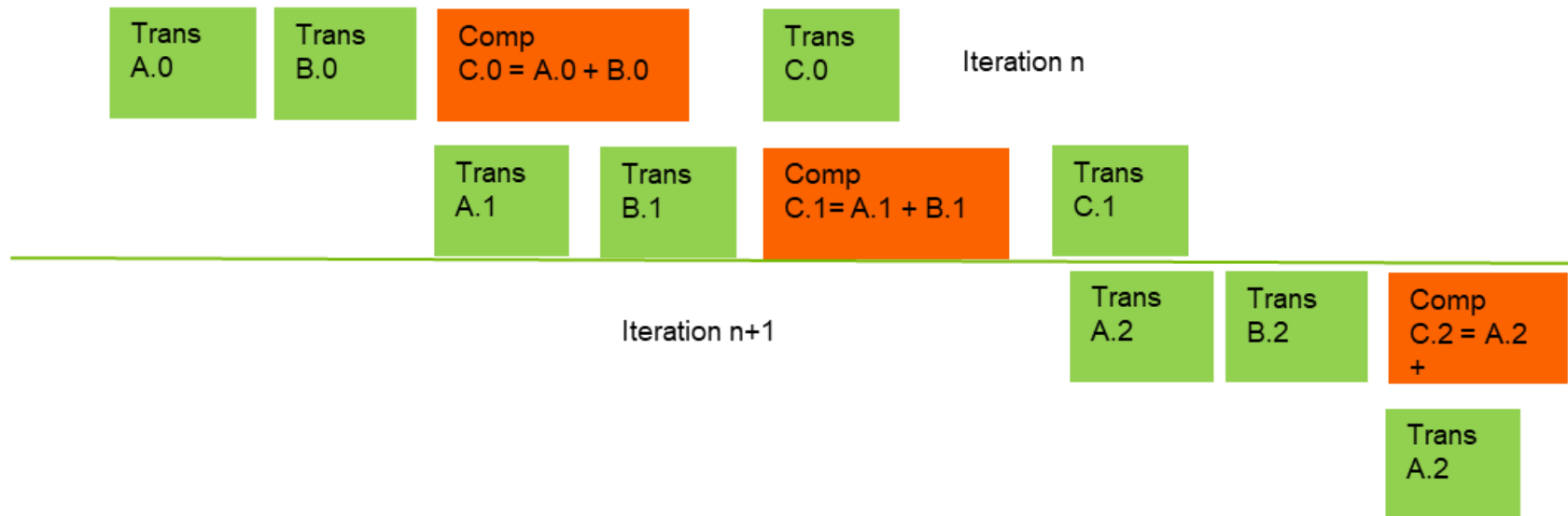


Operations (Kernel launches, cudaMemcpy() calls)

# Better, not quite the best overlap

---

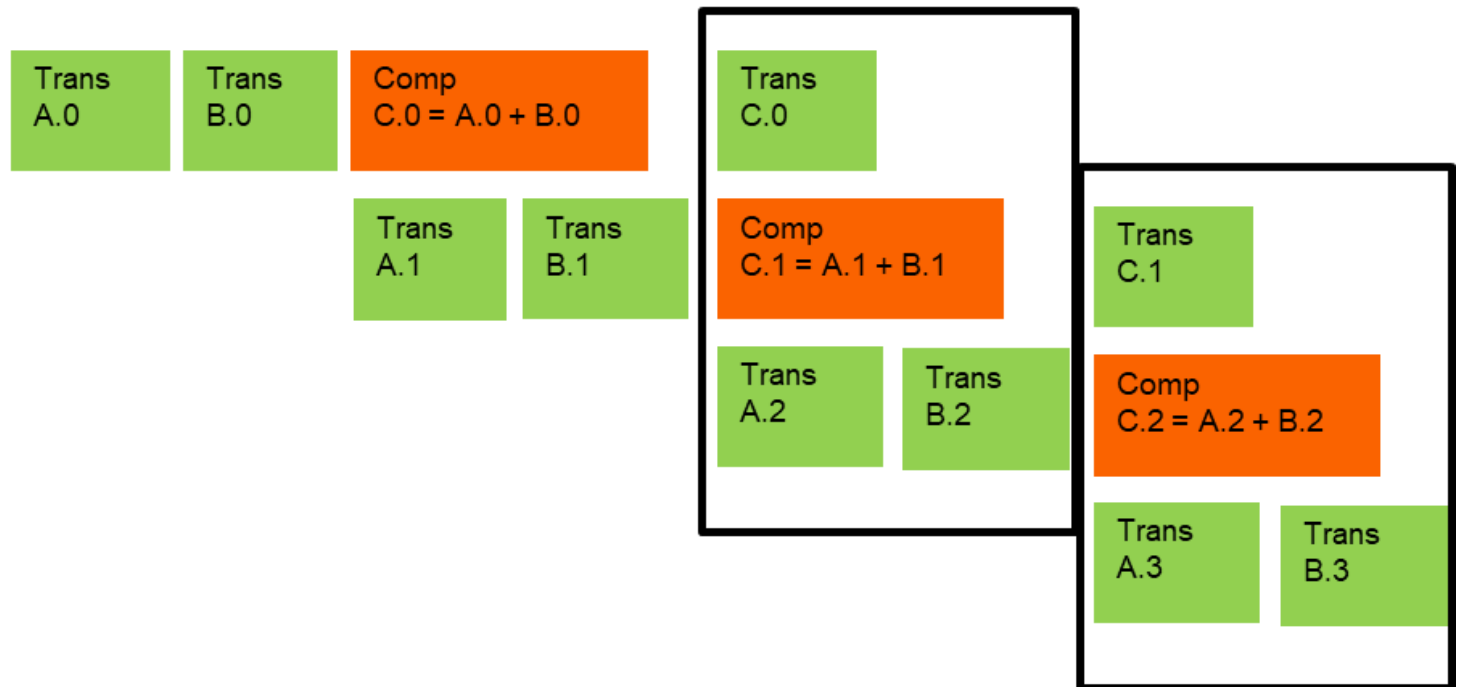
- C.1 blocks next iteration A.0 and B.0 in the copy engine queue



# Ideal, Pipelined Timing

---

- Will need at least three buffers for each original A, B, and C, code is more complicated



# Introduction to CUDA

---

## Questions?

Contact information

Andreas Axelsson

Email: [andreas.axelsson@ju.se](mailto:andreas.axelsson@ju.se)

Mobile: 0709-467760