Computing on GPUs

A HEP practitioner's introduction

Enrico Bothmann, MCnet Summer School 2023, 2023-07-12

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

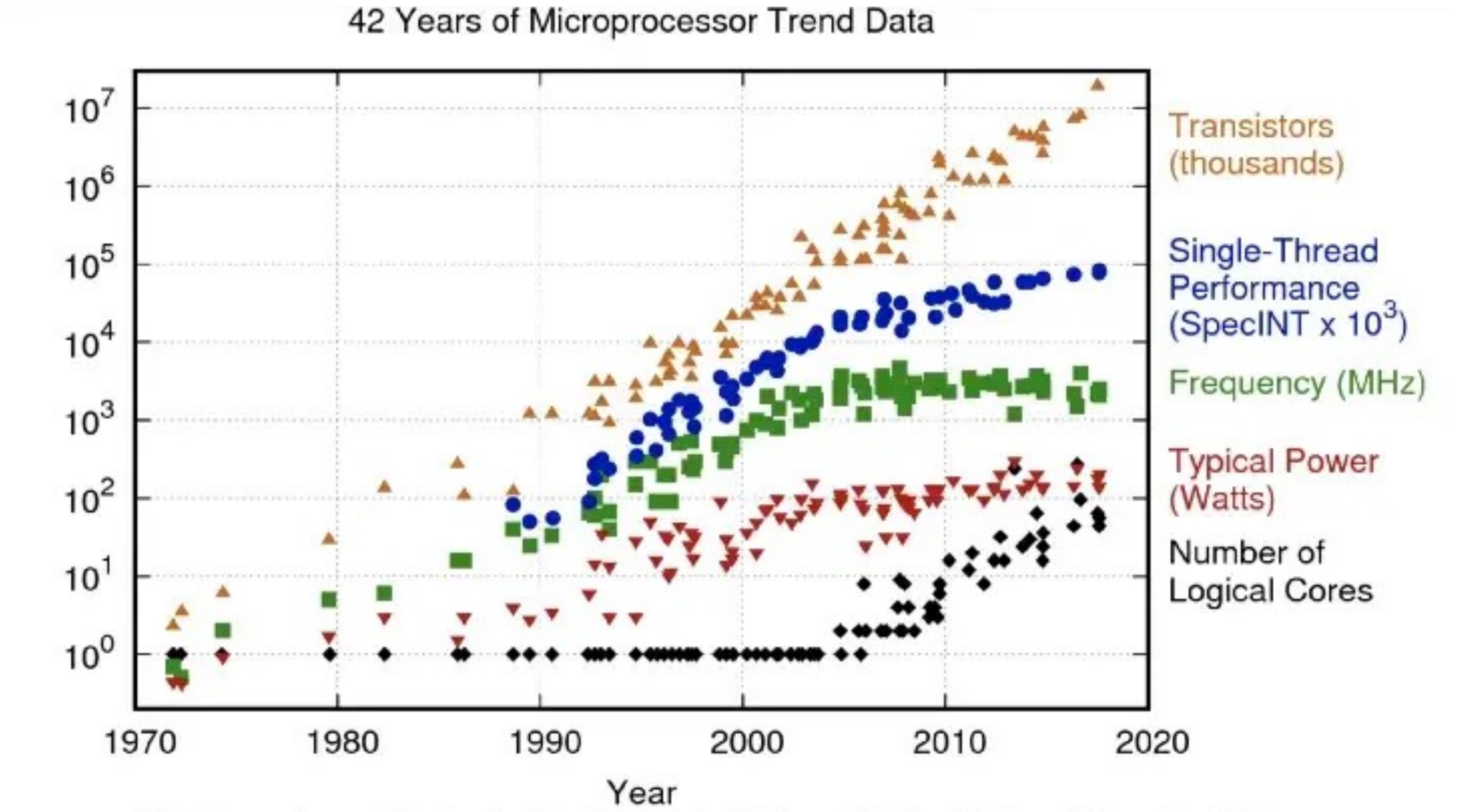
MCnet

Funded by DFG Deutsche Forschungsgemeinschaft German Research Foundation

Contents

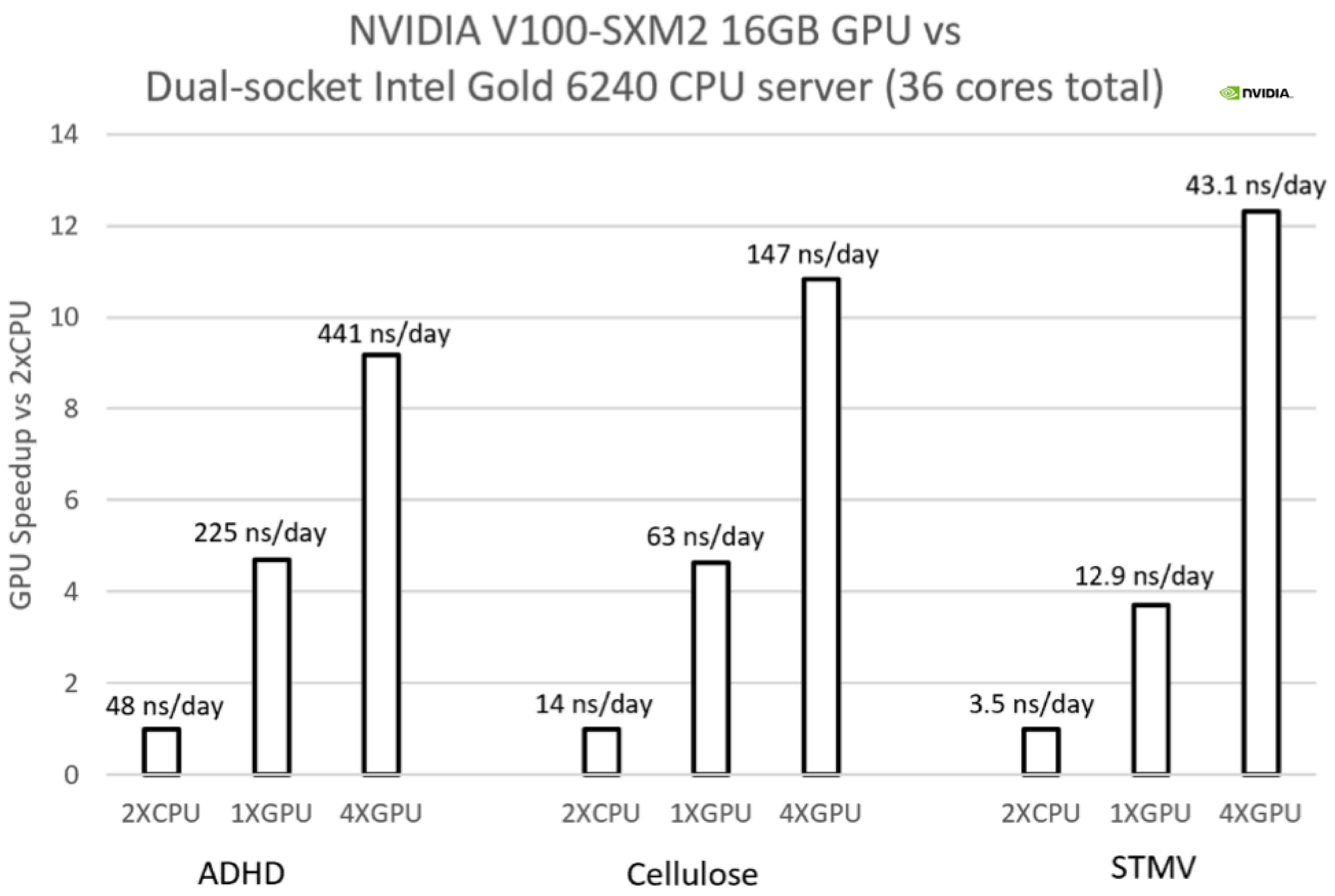
Motivation

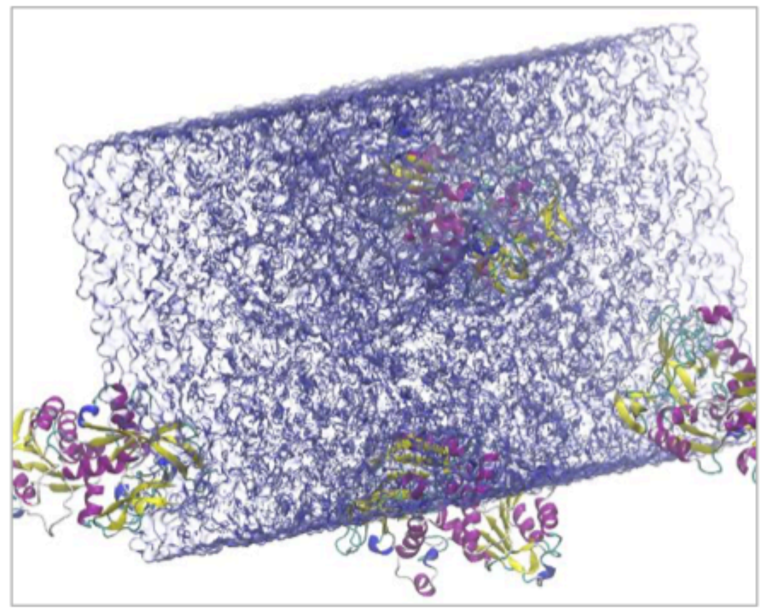
The protracted death of Moore’s law

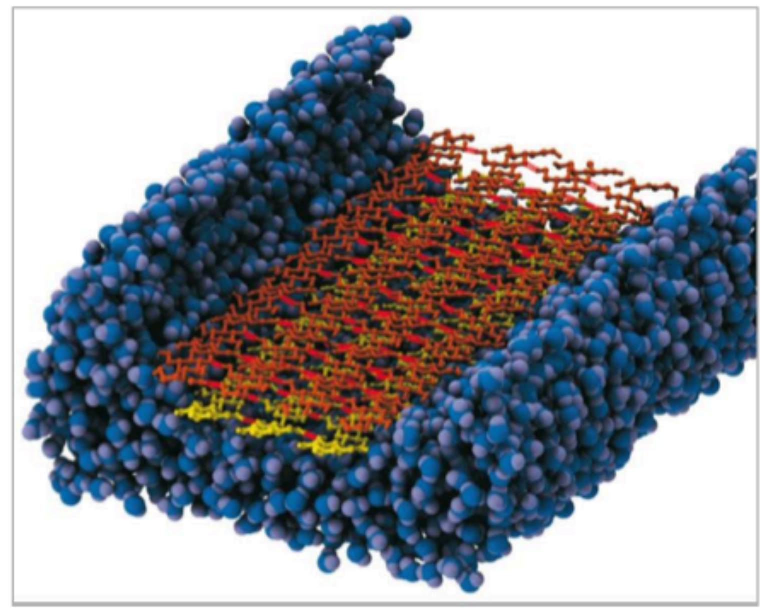


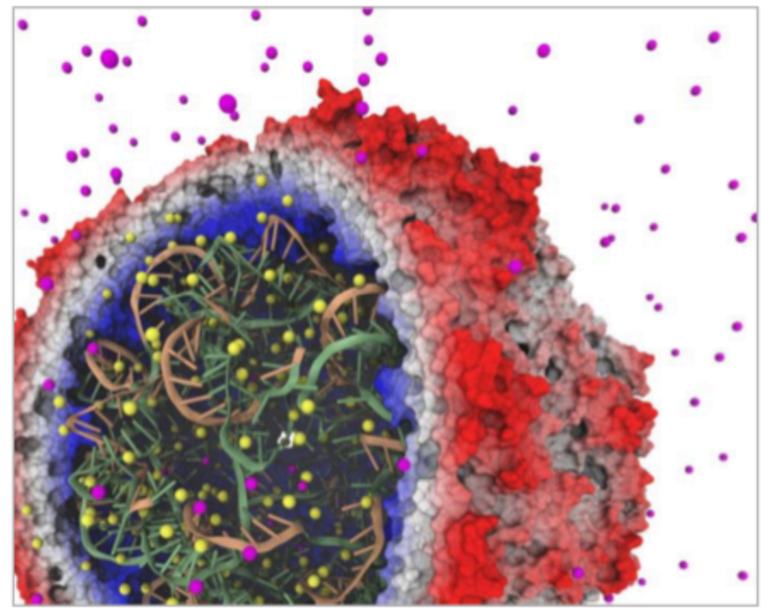
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

GROMACS 2020: GPU VS CPU




ADH Dodec
~100K atoms


Cellulose
~400K atoms


STMV
~1M atoms

Lattice QCD

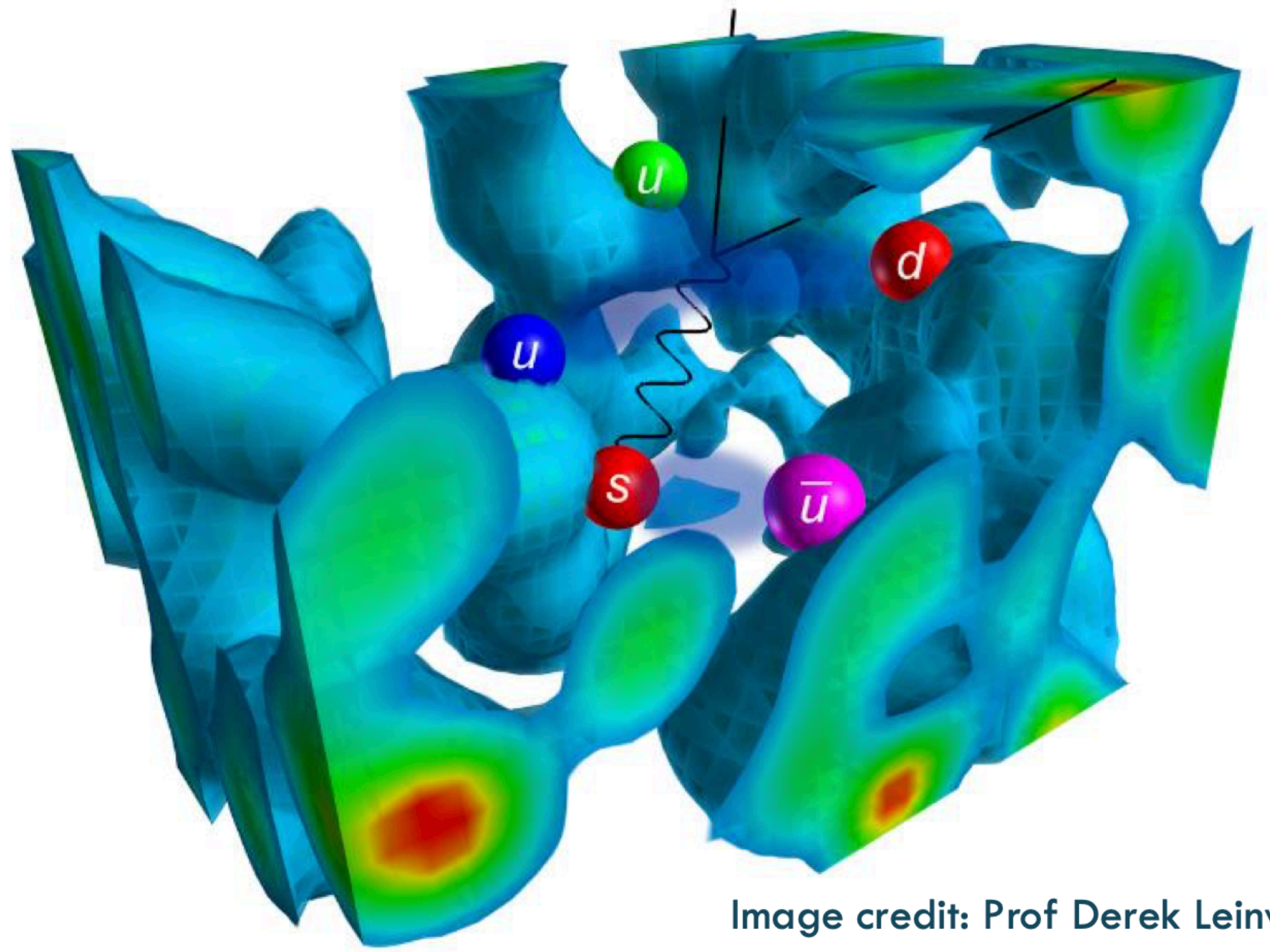
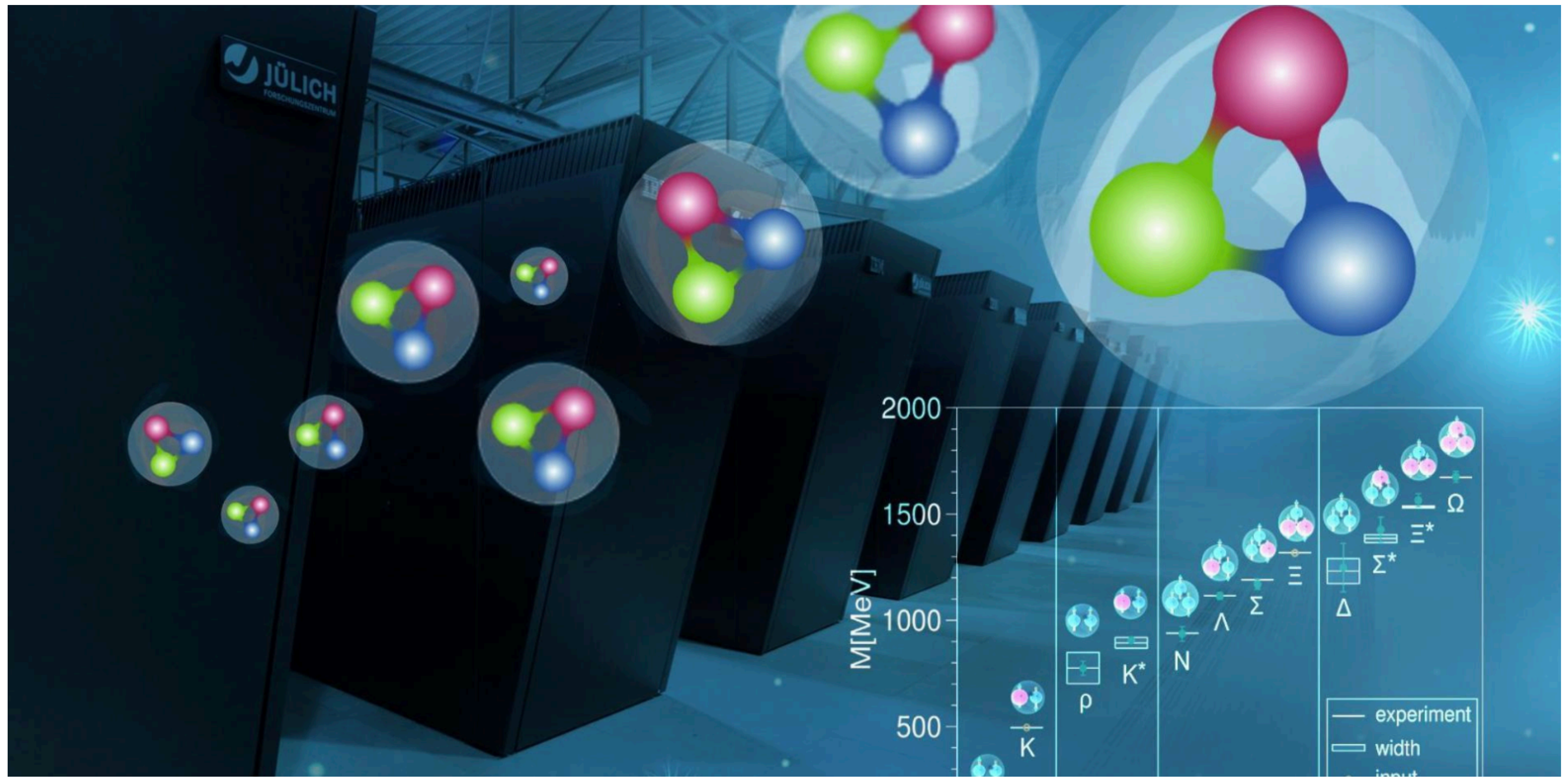


Image credit: Prof Derek Leinweber



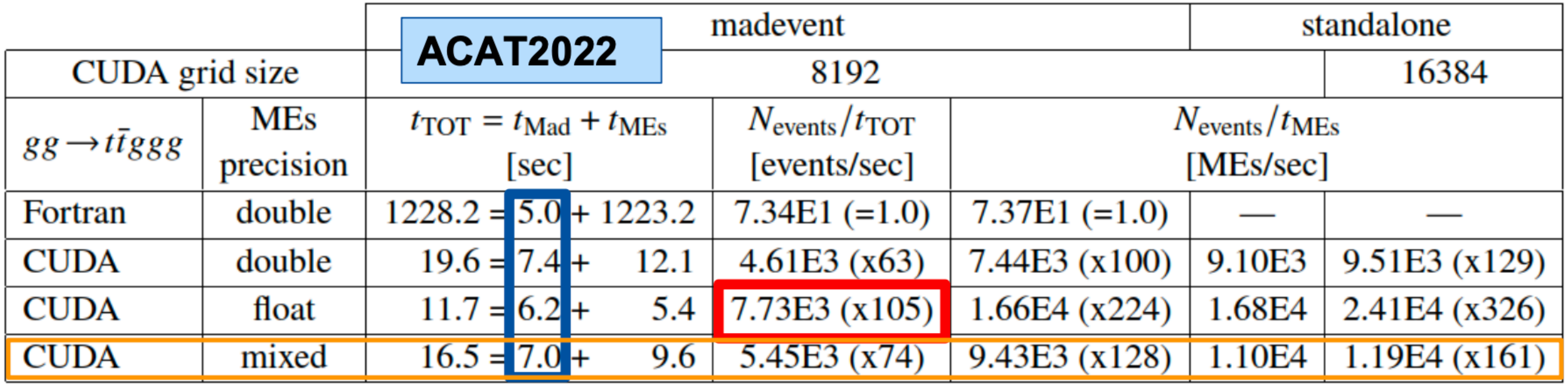
calculate properties of hadrons by simulation on finite four-dimensional lattice with quark and gluon fields

- early adopters of GPU (since ~2006, then 20x speed-up reported)
- Now, cutting edge lattice QCD calculations all use GPU

madgraph4gpu

Porting of MadGraph5_aMC@NLO [Valassi et al. ACAT proceedings (2023) 2303.18244]

- Collaboration of theoretical/experimental physicists with software engineers
- reengineering of MG5aMC is close to a fully functional alpha release for LO QCD processes
- On GPUs, using CUDA, achieve O(100–1000) speedups for MEs alone



ACAT2022

		madevent			standalone	
CUDA grid size		8192				16384
$gg \to t\bar{t}ggg$	MEs precision	$t_{\mathrm{TOT}} = t_{\mathrm{Mad}} + t_{\mathrm{MEs}}$ [sec]	$N_{\mathrm{events}}/t_{\mathrm{TOT}}$ [events/sec]	$N_{\mathrm{events}}/t_{\mathrm{MEs}}$ [MEs/sec]		
Fortran	double	1228.2 = 5.0 + 1223.2	7.34E1 (=1.0)	7.37E1 (=1.0)	—	—
CUDA	double	19.6 = 7.4 + 12.1	4.61E3 (x63)	7.44E3 (x100)	9.10E3	9.51E3 (x129)
CUDA	float	11.7 = 6.2 + 5.4	7.73E3 (x105)	1.66E4 (x224)	1.68E4	2.41E4 (x326)
CUDA	mixed	16.5 = 7.0 + 9.6	5.45E3 (x74)	9.43E3 (x128)	1.10E4	1.19E4 (x161)

PEPPER/SHERPA

New Algorithms for Amplitudes

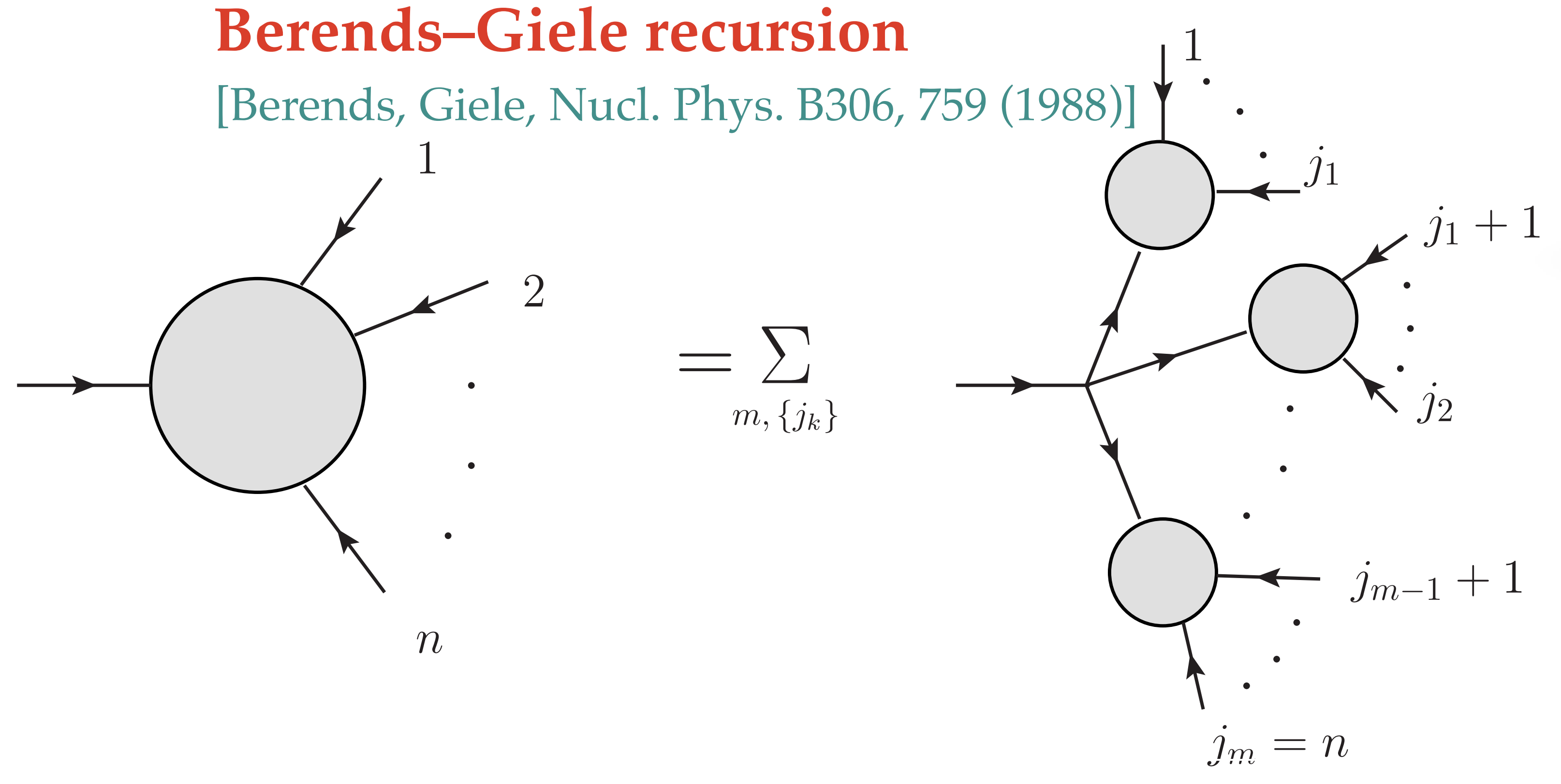
- large-n LO ME and phase-space is bottleneck in state-of-the-art SHERPA production for V+jets & tt+jets
  [EB et al. Eur. Phys. J. C 82 (2022), no. 12, 1128, 2209.00843]
- development of new recursive algorithms that scale well with n (no Feynman diagrams) and are portable/ parallelisable [EB, Giele, Höche, Isaacson, Knobbe, SciPost Phys. Codebases 3 (2022), 2106.06507], [EB et al., submitted to SciPost Phys., 2302.10449]
- most work is done directly on the GPU
  random number generation, sampling, recursive calculation of $|\mathcal{M}|^2$

➡ chip-to-chip speed-ups of $\mathcal{O}(10^1\ldots10^2)$ enable higher precision / study of more complex procs

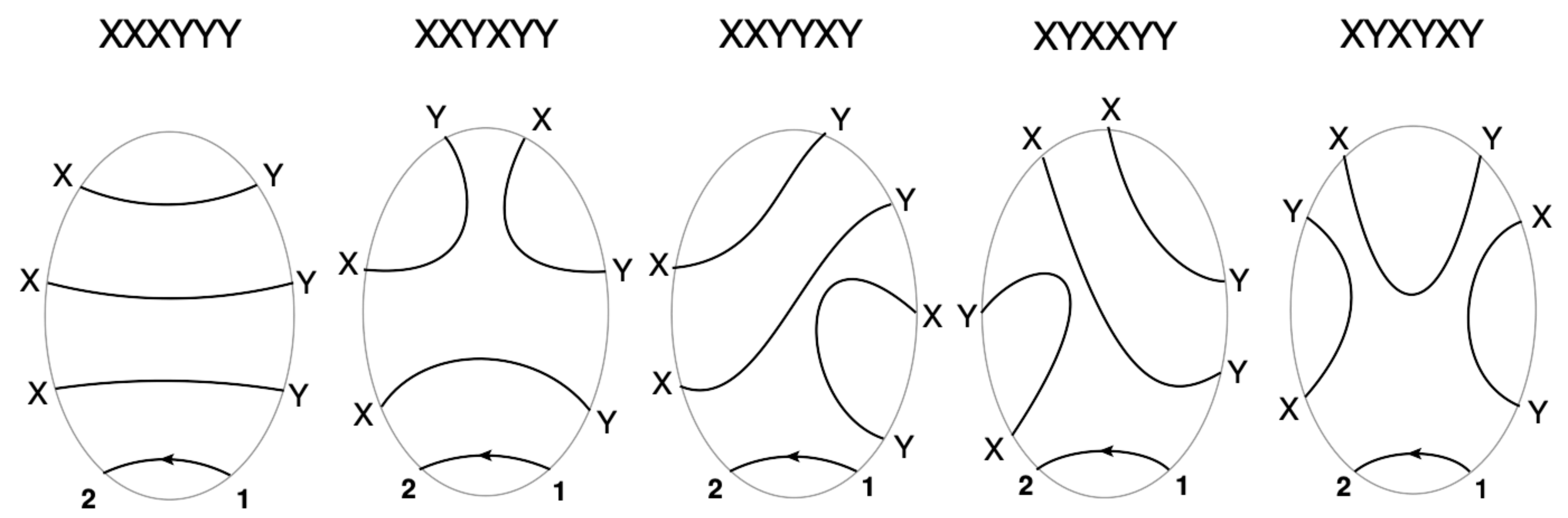
➡ close to public release of fully ported ME generator

Funded by DFG

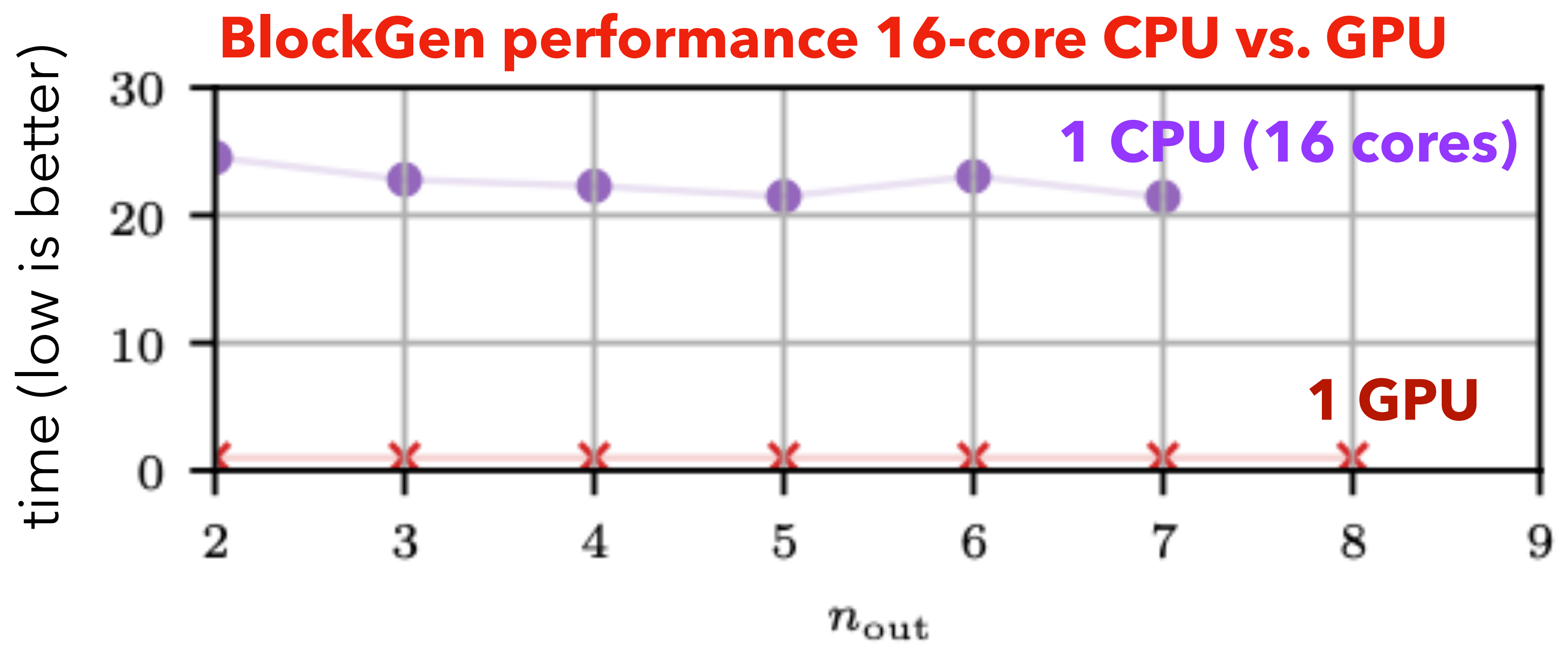
Berends–Giele recursion
[Berends, Giele, Nucl. Phys. B306, 759 (1988)]



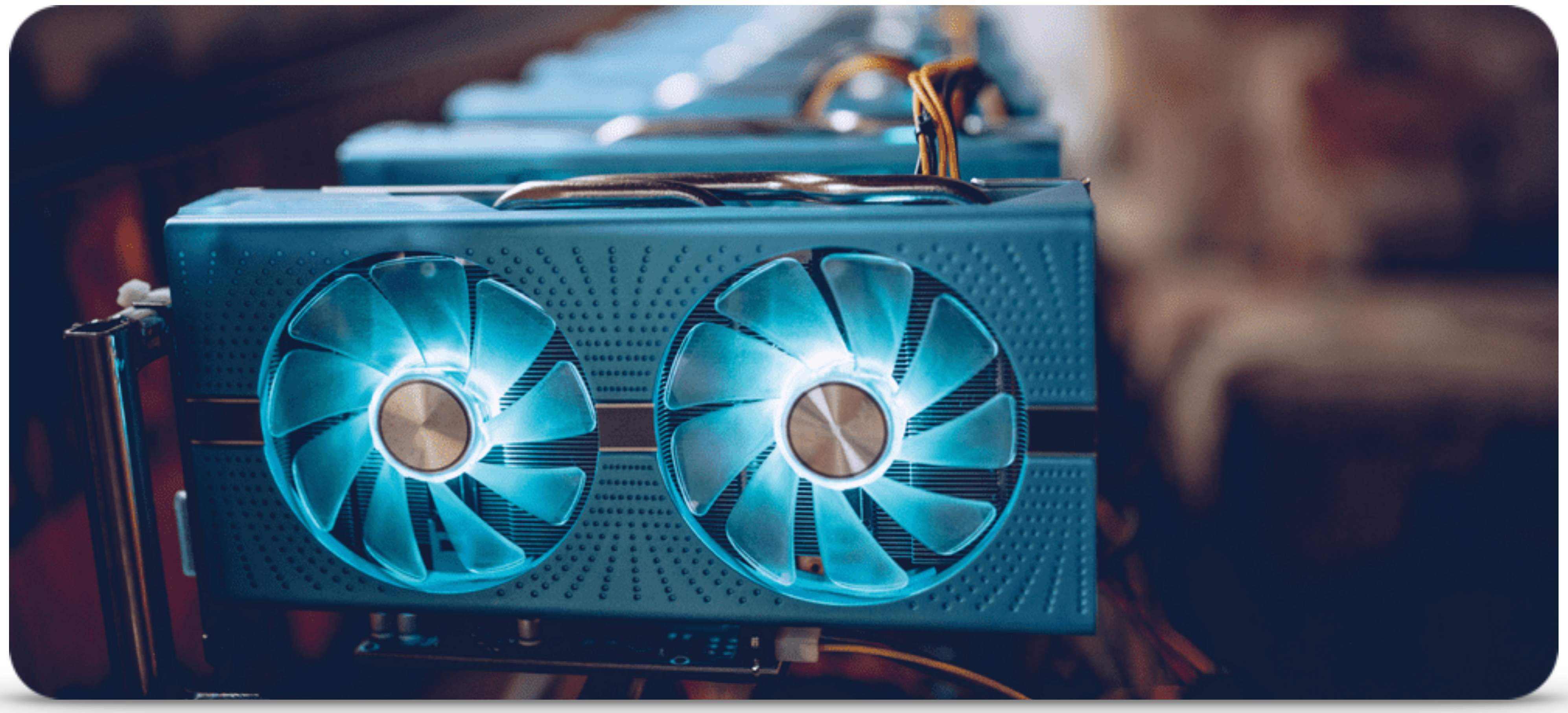
Dyck words of minimal amplitudes basis [Melia 1304.7809]



BlockGen performance 16-core CPU vs. GPU



Motivation



current research including GPU use/porting in HEP

- MCnet: MADGRAPH5_AMC@NLO, PEPPER/SHERPA, ML, …
- experiments: trigger, detector simulation, data compression, ML, …
- NOTE: better performance means increased physics range! (e.g. multiplicity in event generators)

HPC Clusters more and more equipped/built around GPU

- this is a persistent trend since throughput is better for most applications, also fueled by DNN „revolution“
  - 9/10 top supercomputers and 10/10 top „green“ supercomputers use GPU or some other accelerator
  - e.g. SUMMIT cluster (top 5 supercomputer) 95 % of compute capacity via GPU
- porting our tools would allow better exploitation of HPC resources now & in the future

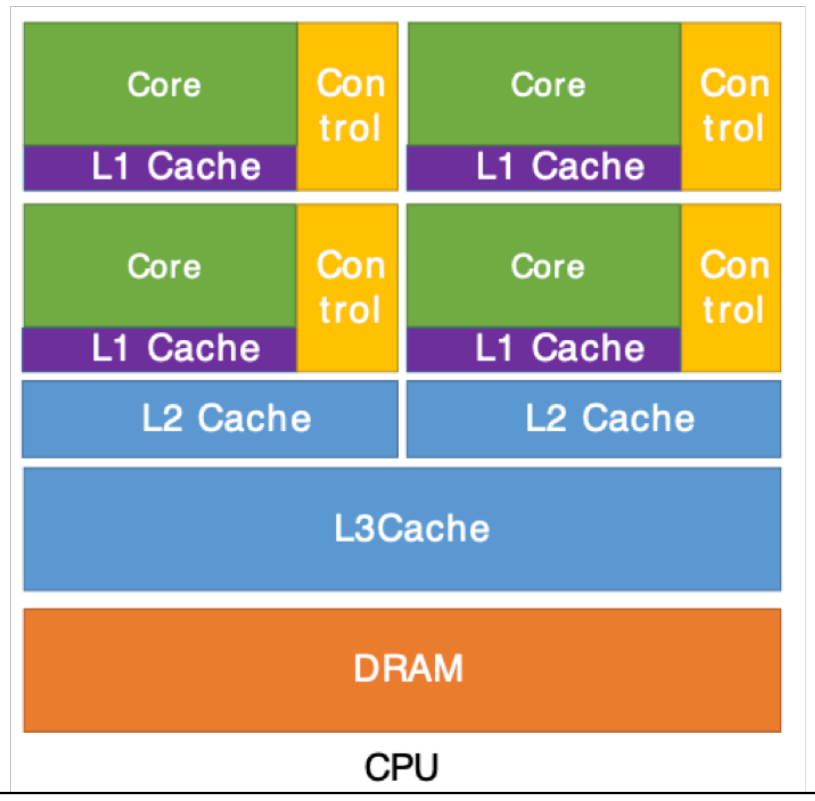
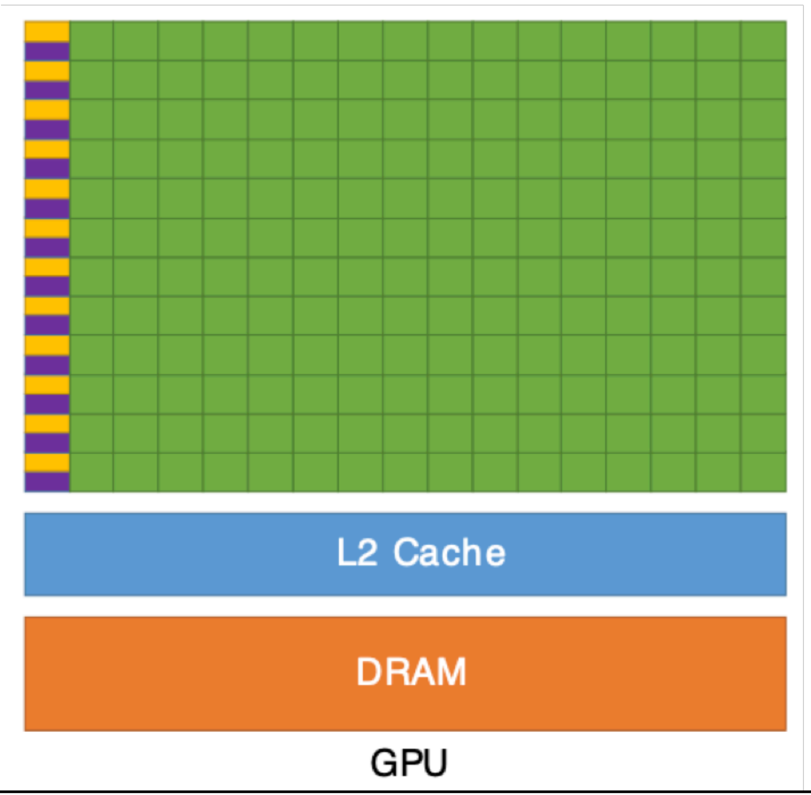
CPU vs. GPU efficiencies

Theoretical comparison on a development node of ITP Göttingen

	€	MFLOPS (double precision)	Watt	€ / MFLOPS	MFLOPS / Watt
Intel®Xeon®Silver 4214R 2.4 GHz 12core; 192 GB	800	40 (single thread: 2)	100	20	0.4
NVIDIA® Tesla V100S 32 GB	6000	8200	250	0.7	32.8

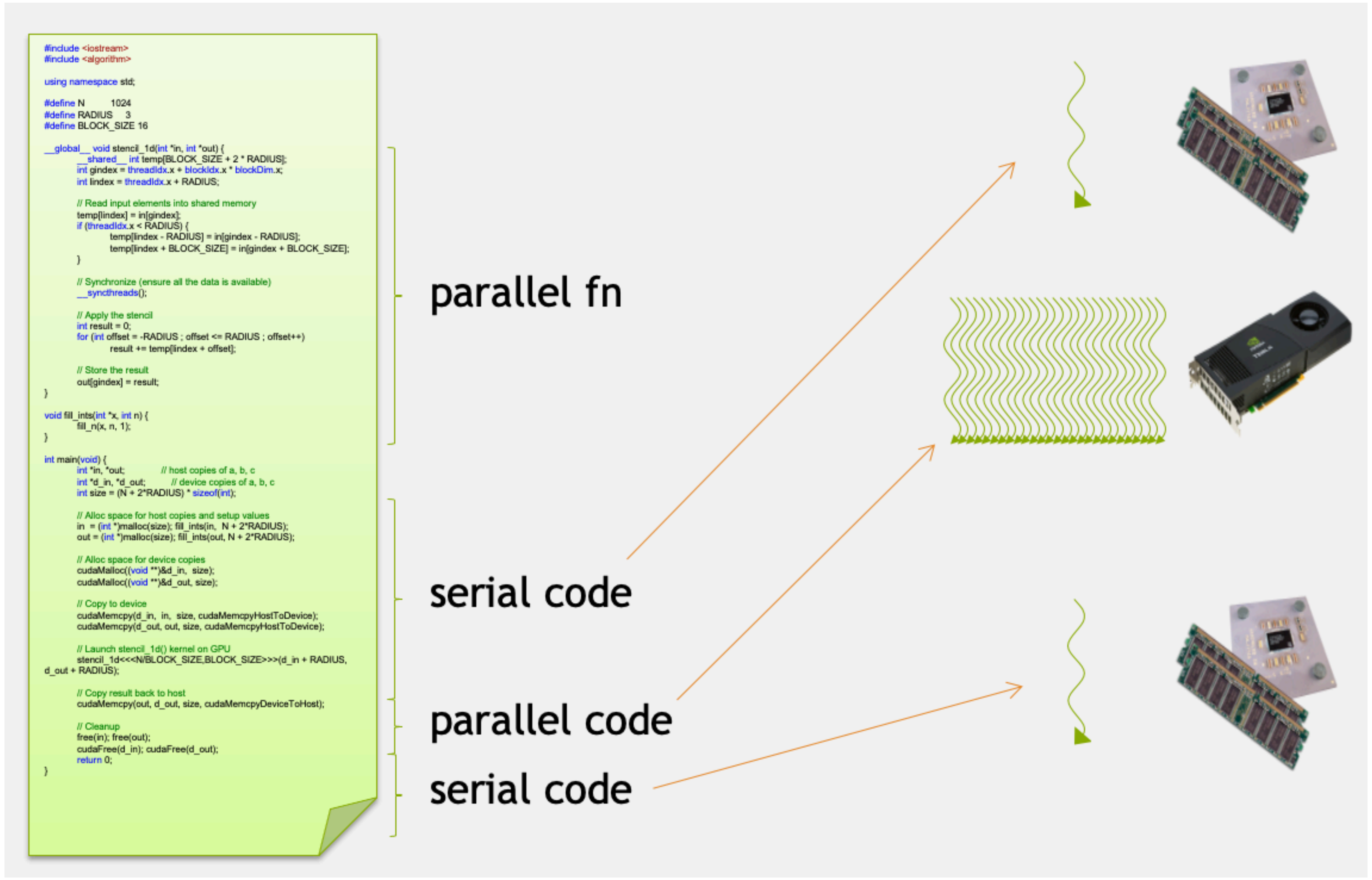
Heterogenous Programming

Heterogeneous programming

	CPU	GPU
optimised for …	racing through serial operations	doing massively parallel calculations
minimises …	latency (= how long it takes to finish a task once)	throughput (= how many times a task is completed within a time period)
architecture	high clock speed, lots of control logic, large caches, fast memory  Core Control L1 Cache L2 Cache L3Cache DRAM CPU	lots of transistors dedicated to calculations, large memory bandwidth 
for HEP applications	good for complex algorithms with lots of branching (if/else) and lots of caching	good for comparably simple algorithms which are easy to parallelise (prime example: Monte Carlo)

Heterogeneous programming

CPU + GPU: host/device programming model

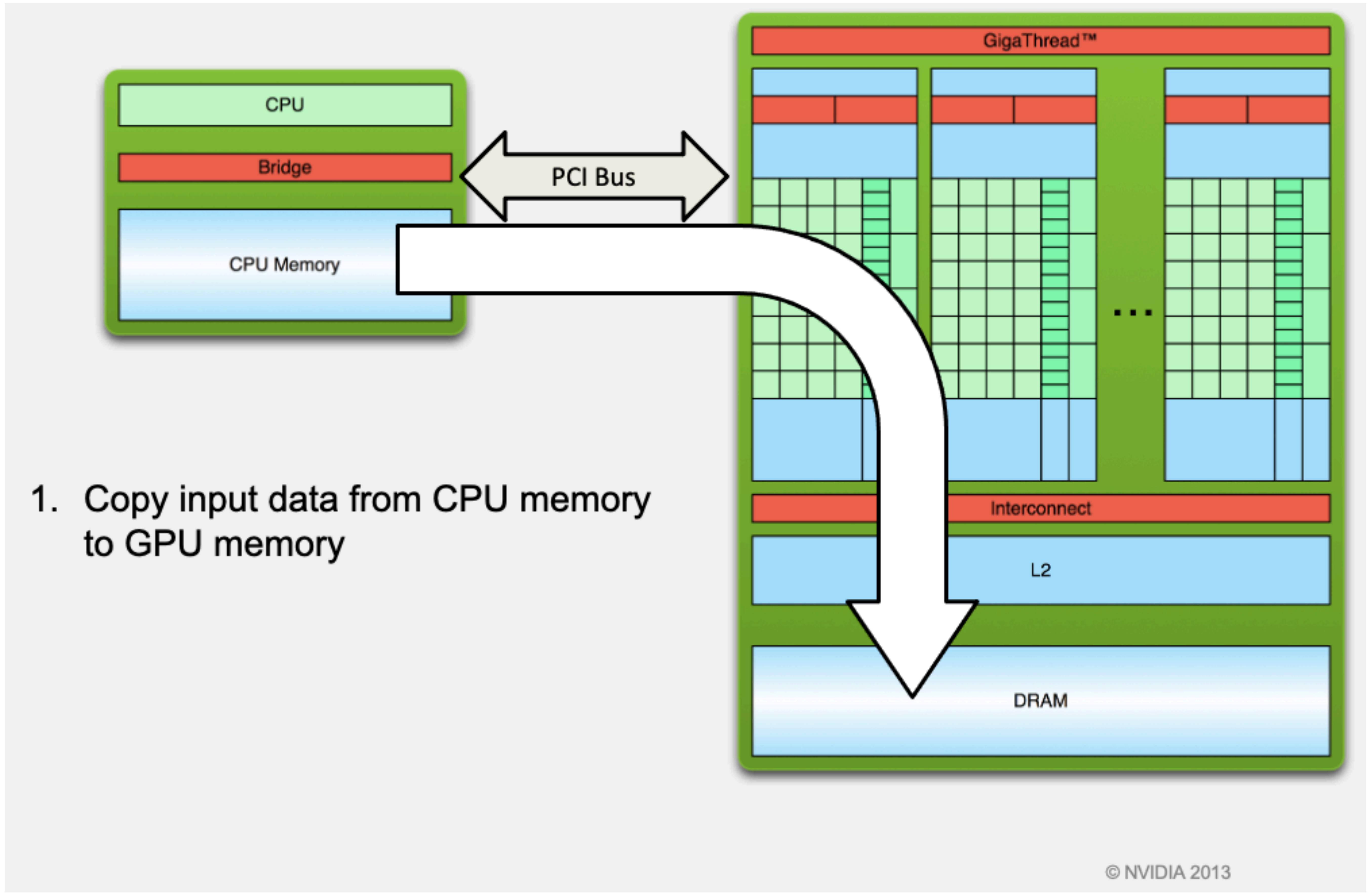


```
#include <iostream>
#include <algorithm>

using namespace std;

#define N        1024
#define RADIUS    3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;           // host copies of a, b, c
        int *d_in, *d_out;       // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

- simplify porting by only parallelising the parts of the code that matter („hot spots“)
- complicated by additional time needed to copy data between host and device memory

Heterogeneous programming

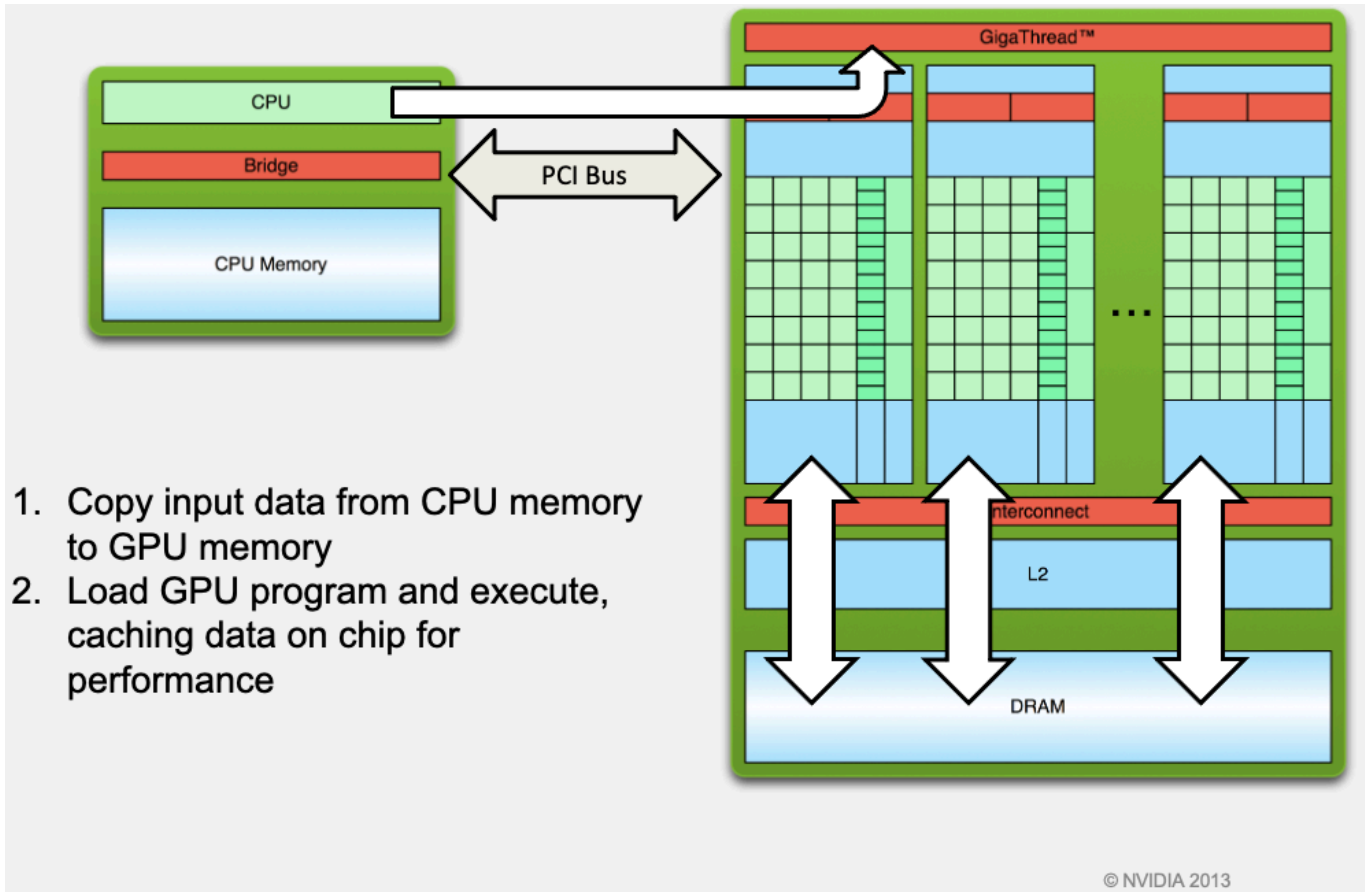
Simple processing flow



1. Copy input data from CPU memory to GPU memory

Heterogeneous programming

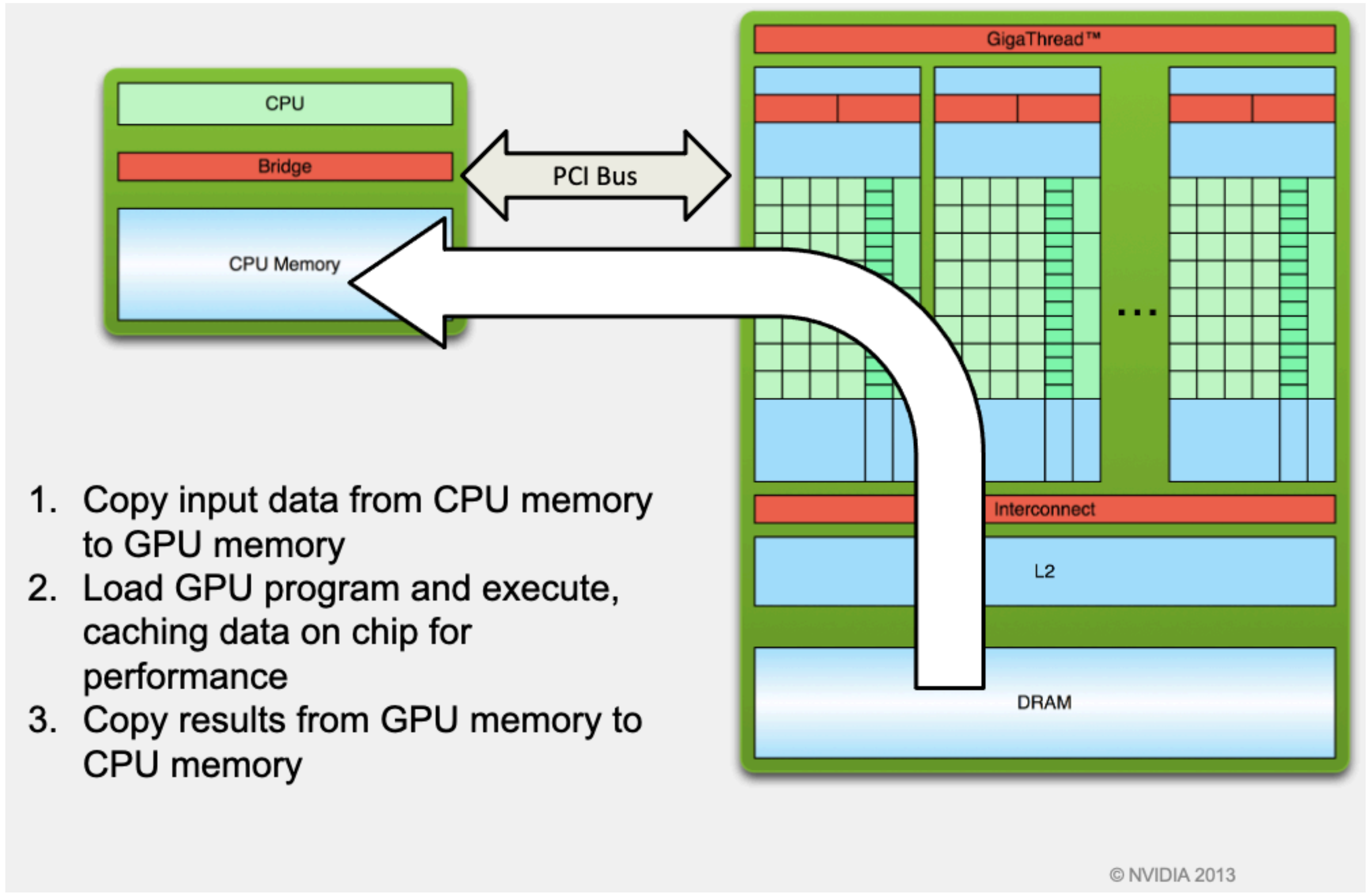
Simple processing flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Heterogeneous programming

Simple processing flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Introduction to CUDA

Introduction to CUDA programming

Software/Hardware

- CUDA: compiler (nvcc), C language extensions and libraries by NVidia to run calculations on their GPU
- Can I use CUDA on any (NVidia) GPU?
  - No … and for many scientific codes you might need a *data center* GPU:

	example	**32FP TFLOPS**	**64FP TFLOPS**
gaming	NVidia GeForce RTX 3090	35.6	0.556
data center	NVidia V100S	16.4	8.2

Introduction to CUDA programming

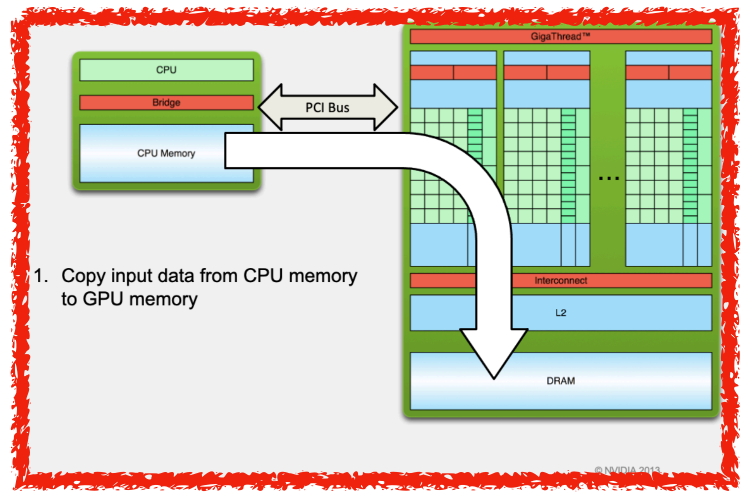
Mixing host and device code

```
__global__ void compute(void) {
  // this runs parallelised over BxT threads on the GPU
}

int main(void) {
  compute<<<B, T>>>();
  printf("Hello World!\n");
  return 0;
}
```

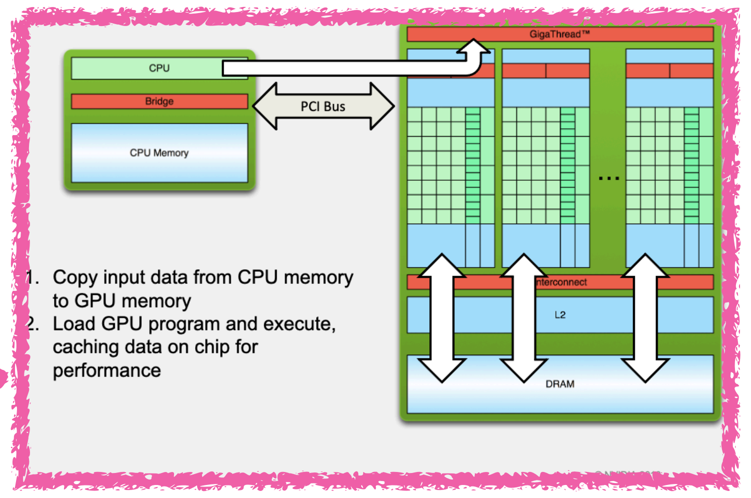
- `__global__`: function callable by host, executed on device
- `<<<B, T>>>(…)`: call from host code to device code
  - B: number of thread blocks
  - T: number of threads per block

Complete vector addition example

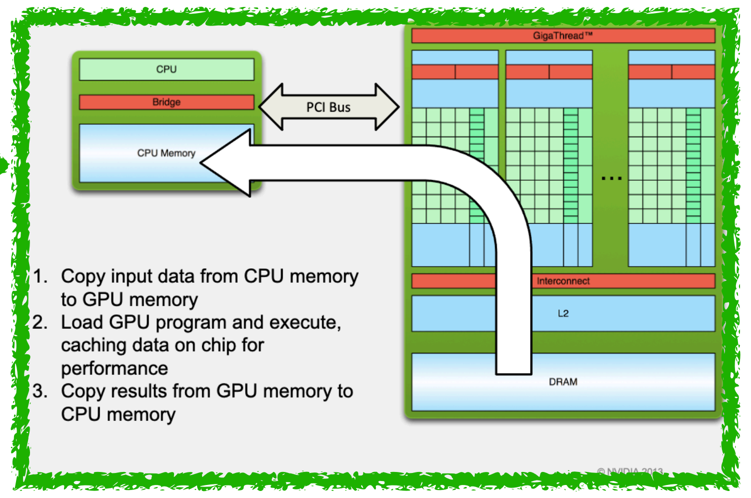
```
__global__ void add(int *a, int *b, int *c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}

int main(void) {
  int size = N * sizeof(int);

  // host memory for vectors a, b, c
  int *a, *b, *c;
  a = malloc(size); b = malloc(size); c = malloc(size);
  random_fill(a);
  random_fill(b);

  // device memory for vectors a, b, c
  int *da, *db, *dc;
  cudaMalloc(*da, size); cudaMalloc(*db, size); cudaMalloc(*dc, size);

  // copy a, b to device
  cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

  compute<<<B, T>>>(a, b, c);

  // copy c from device
  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

  free(a); free(b); free(c);
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

  return 0;
}
```



1. Copy input data from CPU memory to GPU memory



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Introduction to CUDA programming

For a C++ like approach, use CUDA's thrust library:
(will use a mix of pure CUDA and Thrust API in the tutorial)

```
int main(void) {
  // host memory for vectors a, b, c
  thrust::host_vector<int> a(N), b(N), c(N);

  // device memory for vectors a, b, c
  thrust::device_vector<int> d_a(N), d_b(N), d_c(N);

  random_fill(a);
  random_fill(b);

  // copy a, b to device
  d_a = a;
  d_b = b;

  thrust::transform(d_a.begin(), d_a.end(),
                    d_b.begin(),
                    d_c.begin(), thrust::plus<int>());

  // copy c from device
  c = d_c;

  return 0;
}
```

Examples also exist in the Python world:

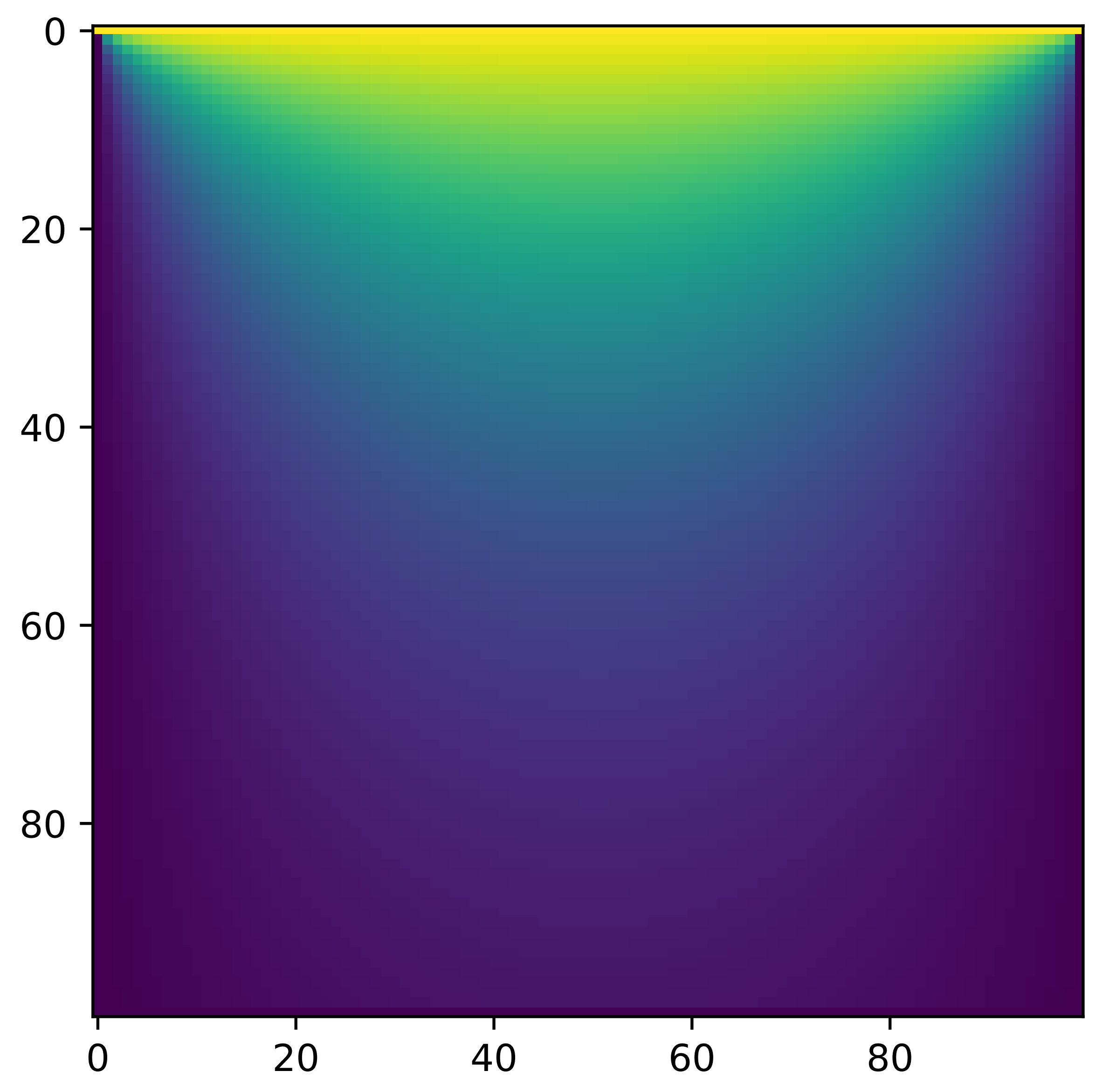
```
from numba import cuda
import numpy as np

@cuda.jit
def add(a, b, c):
  i = (cuda.blockIdx.x * cuda.blockDim.x
       + cuda.threadIdx.x)
  c[i] = a[i] + b[i]

a = np.random(N)
b = np.random(N)
c = np.zeros(N)

# numpy array copied automatically
add(a, b, c)
```

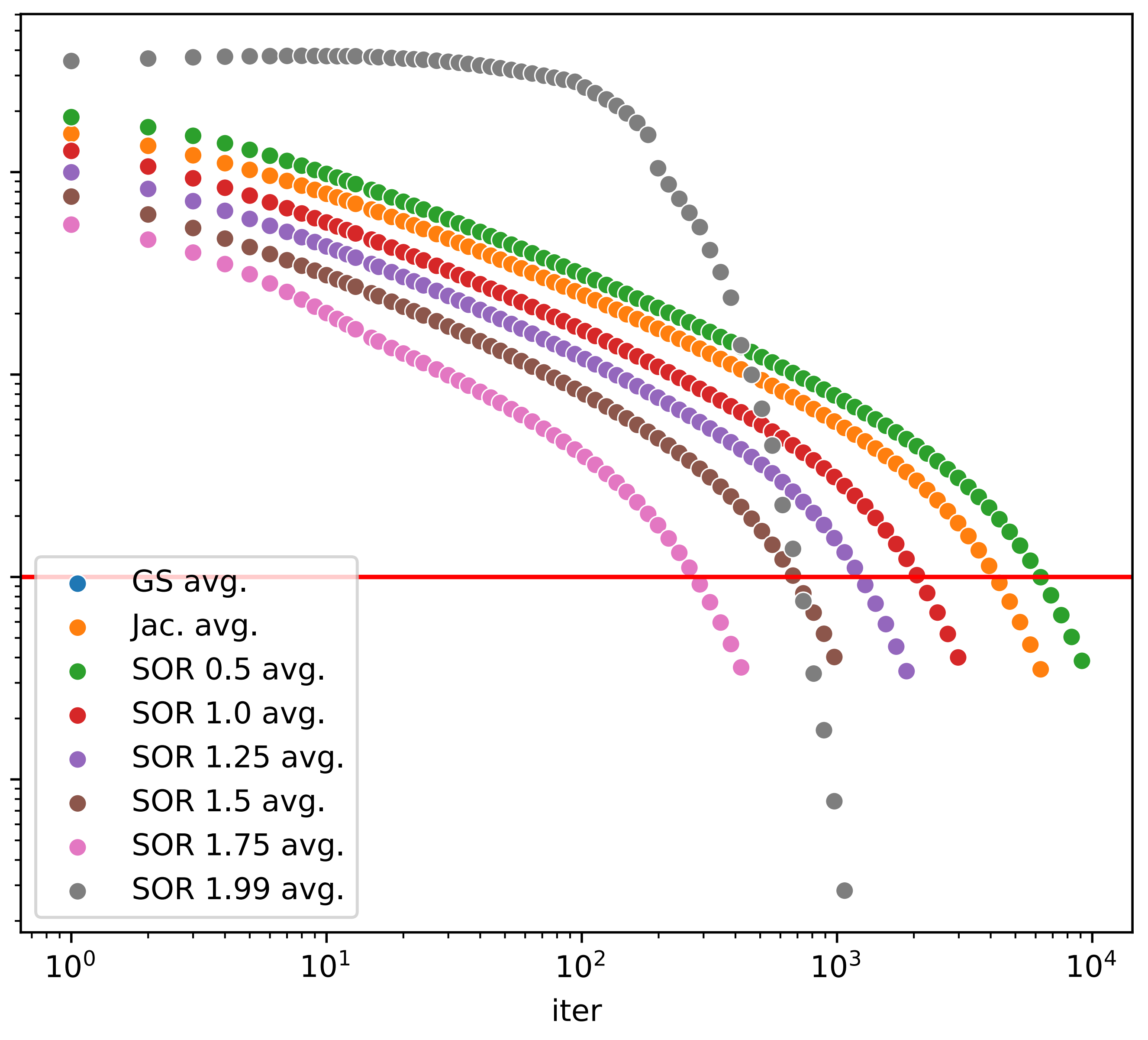
or use pyCUDA, pyTorch, TensorFlow …

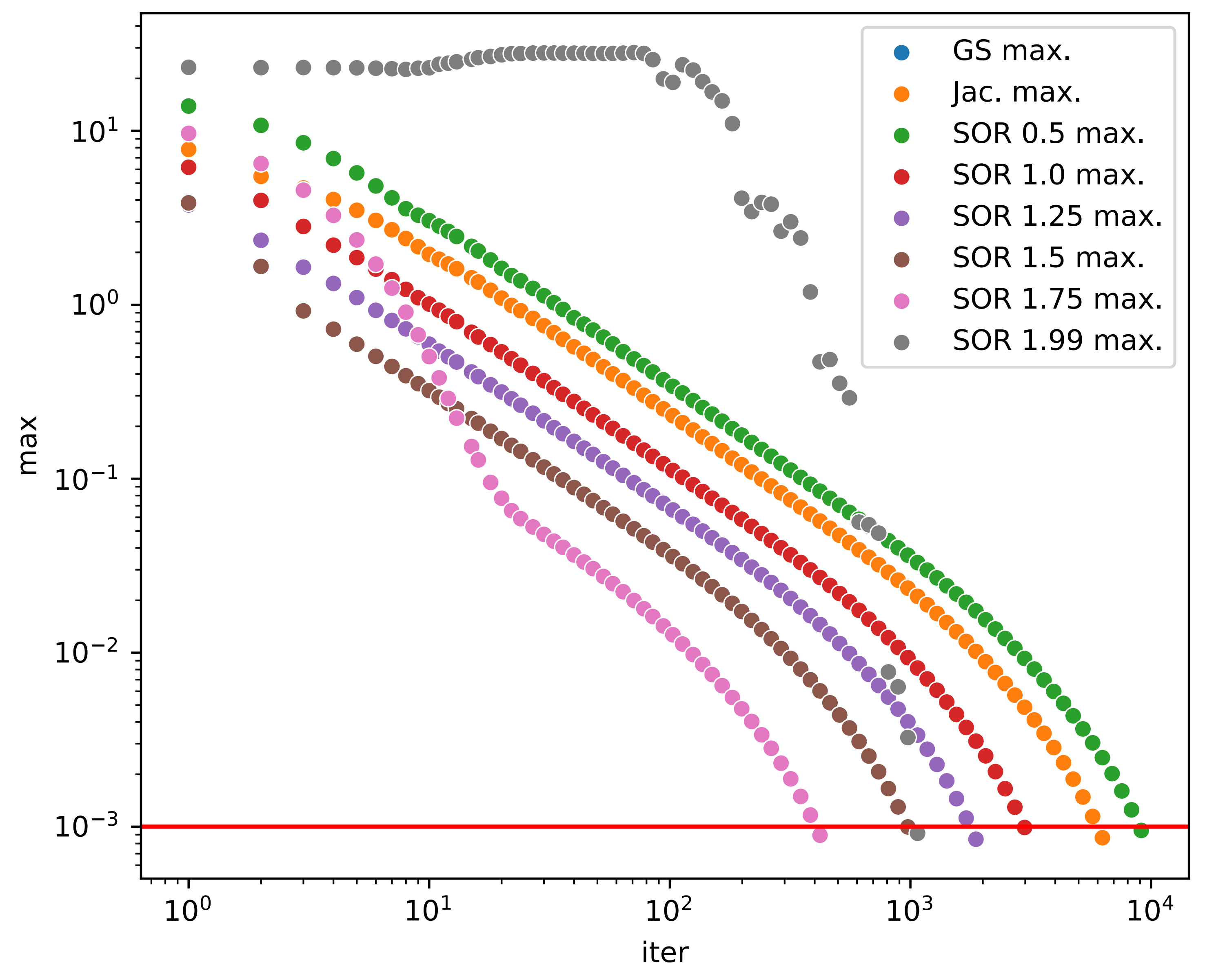
Example: Jacobi Method

Example: Jacobi method

task: solve $\Delta\phi(x,y) = 0$ iteratively on an NxN grid, Dirichlet boundary conditions





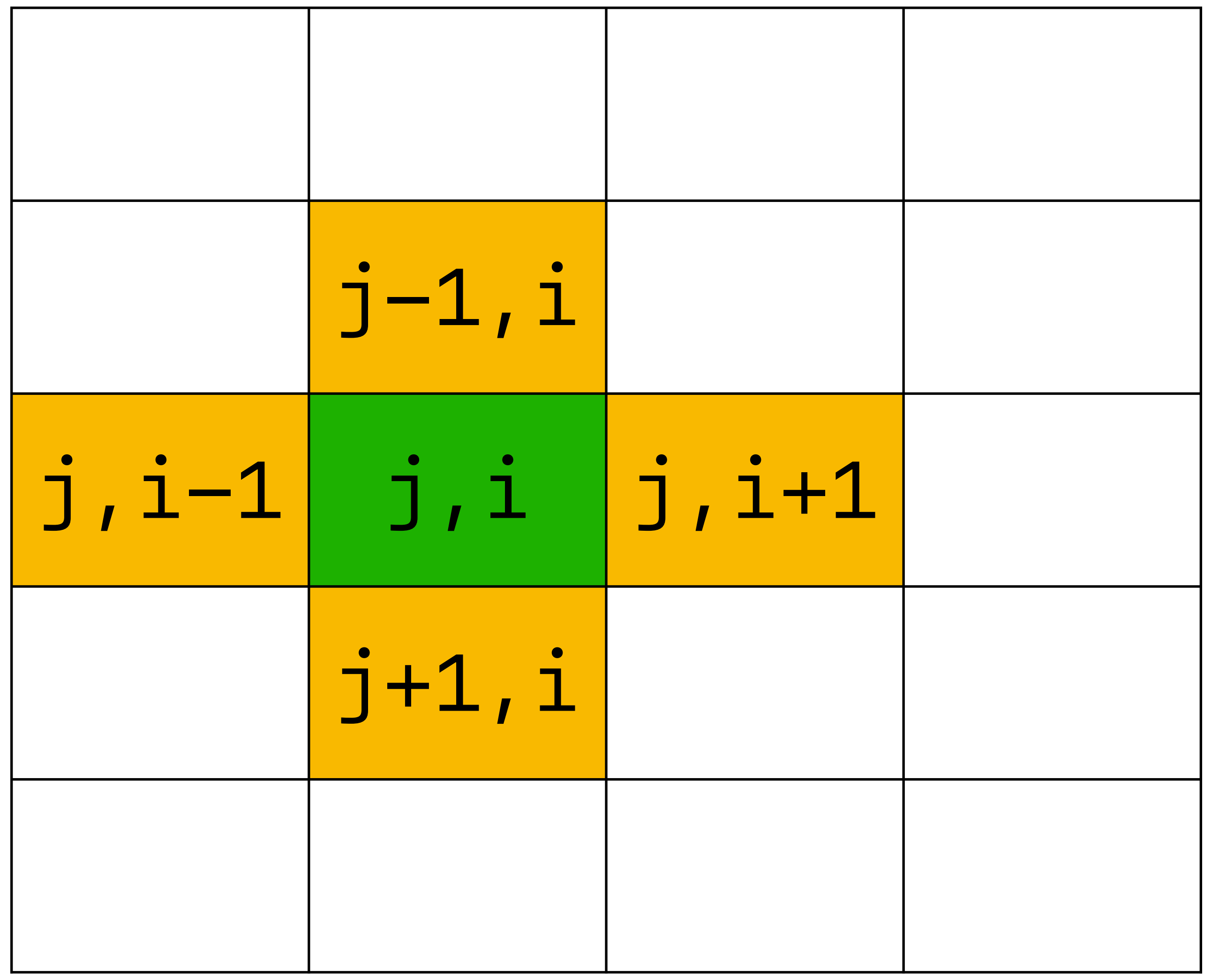


Example: Jacobi method

Review: Algorithm

- set each point to average of neighbours:

```
phi[j,i] = 0.25 * (phi_prev[j,i-1]
                 + phi_prev[j,i+1]
                 + phi_prev[j-1,i]
                 + phi_prev[j+1,i] - 4 * h * h)
```



- `phi_prev` = values from previous iteration
- h = grid spacing
- repeat for many `Niterations` until converged

Example: Jacobi method

Review: Pseudo Code

```
//Initialization
for j = 1,Ny
  for i = 1,Nx
     phi_prev[j,i] = 0.0

//Boundary Conditions
for i = 0,Nx-1
{
  x = i*h
  phi_prev[0,i]    = x*x
  phi_prev[Nj-1,i] = x*x + 1.0
}
for j = 1,Ny-2
{
  y = j*h
  phi_prev[j,0]    = y*y
  phi_prev[j,Ni-1] = 1.0 + y*y
}
```

```
//Iteration Loop
for k = 1,Niterations
{
  for j = 1,Ny-2
    for i = 1,Nx-2
      phi[j,i] = 0.25 * (
            phi_prev[j,i-1]
          + phi_prev[j,i+1]
          + phi_prev[j-1,i]
          + phi_prev[j+1,i] - 4*h*h)

  for j = 1,Ny-2
    for i = 1,Nx-2
      phi_prev[j,i] = phi[j,i];
}
```

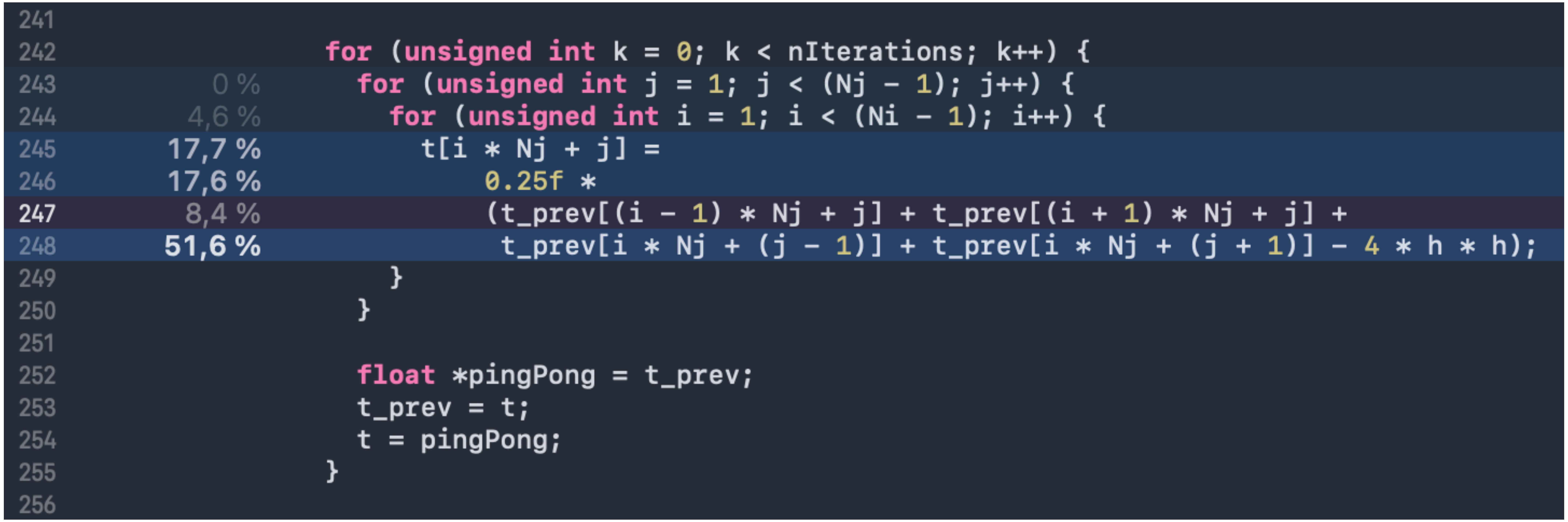
Example: Jacobi method

CPU Time Profiling

Demo

Example: Jacobi method

CPU Time Profiling

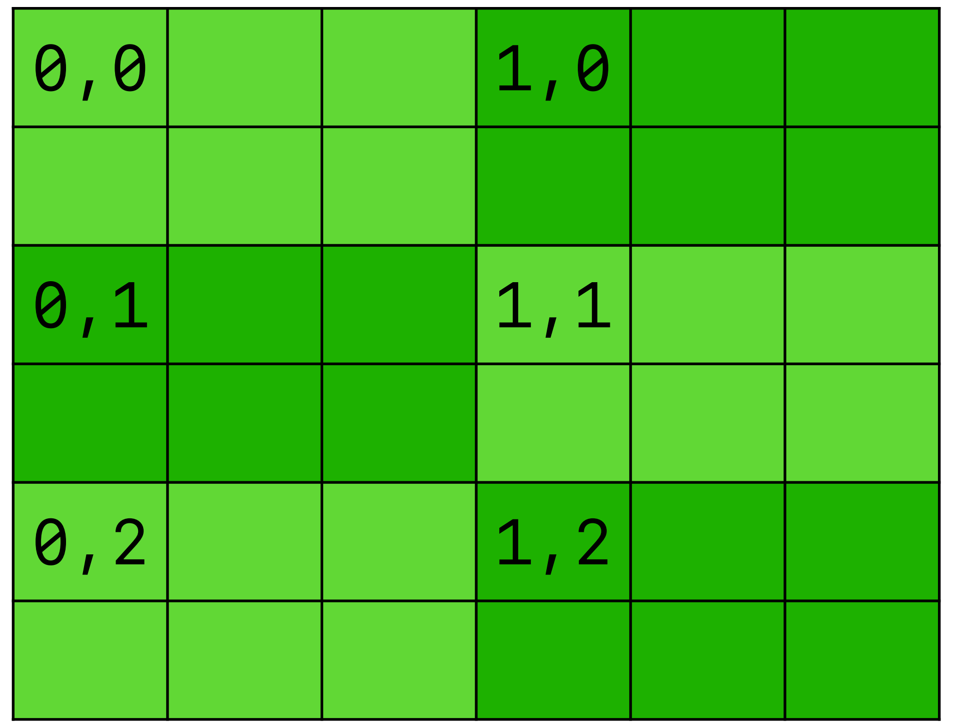


```
                 for (unsigned int k = 0; k < nIterations; k++) {
      0 %          for (unsigned int j = 1; j < (Nj - 1); j++) {
    4,6 %            for (unsigned int i = 1; i < (Ni - 1); i++) {
   17,7 %              t[i * Nj + j] =
   17,6 %                  0.25f *
    8,4 %                  (t_prev[(i - 1) * Nj + j] + t_prev[(i + 1) * Nj + j] +
   51,6 %                   t_prev[i * Nj + (j - 1)] + t_prev[i * Nj + (j + 1)] - 4 * h * h);
                     }
                   }

                   float *pingPong = t_prev;
                   t_prev = t;
                   t = pingPong;
                 }
```

Example: Jacobi method

GPU port: loop over i, j becomes __global__ function

```
  for (int k = 0; k < nIterations; k++) {
    // Launch a kernel on the GPU with one thread for each element.
    calculateJacobi_V1<<<dimGrid, dimBlock>>>(d_phi_prev, d_phi, Nx, Ny, h);

    float *pingPong = d_phi_prev;
    d_phi_prev = d_phi;
    d_phi = pingPong;
  }
```



```
__global__ void calculateJacobi_V1(float *input, float *output,
                                   int Nx, int Ny, float h) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  int j = blockDim.y * blockIdx.y + threadIdx.y;

  if (i > 0 && j > 0 && i < (Nx - 1) && j < (Ny - 1)) {
    float phi = - h * h;
    phi += 0.25f*input[j * Nx + i - 1];
    phi += 0.25f*input[(j - 1) * Nx + i];
    phi += 0.25f*input[j * Nx + i + 1];
    phi += 0.25f*input[(j + 1) * Nx + i];
    output[j * Nx + i] = phi;
  }
}
```

Example: Jacobi method

Port to GPU

Demo

Example: Jacobi method

Port to GPU

```
$  ./jacobi_gpu
Parameters: Nx=2048, Ny=2048, nIteration=1000
CPU Time: 5770.000000 ms
GPU Time: 40.599007 ms
Is host equal to device = 1
Speedup = 142.121704x
All done
```

Pretty good

But far away from the factor one would have hoped for by comparing theoretical FLOPS!

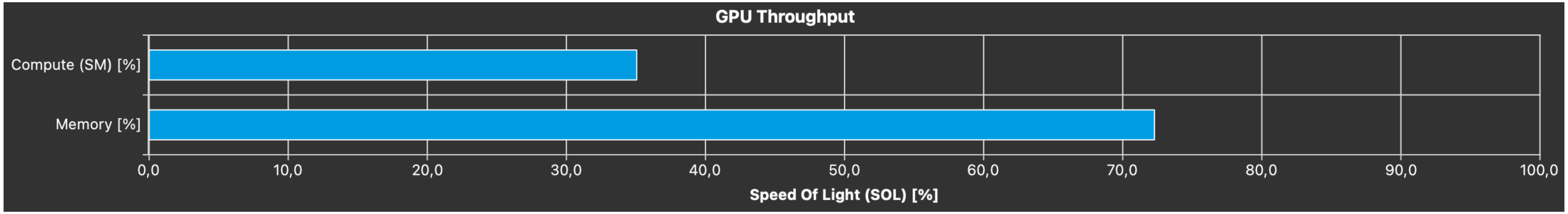
Example: Jacobi method

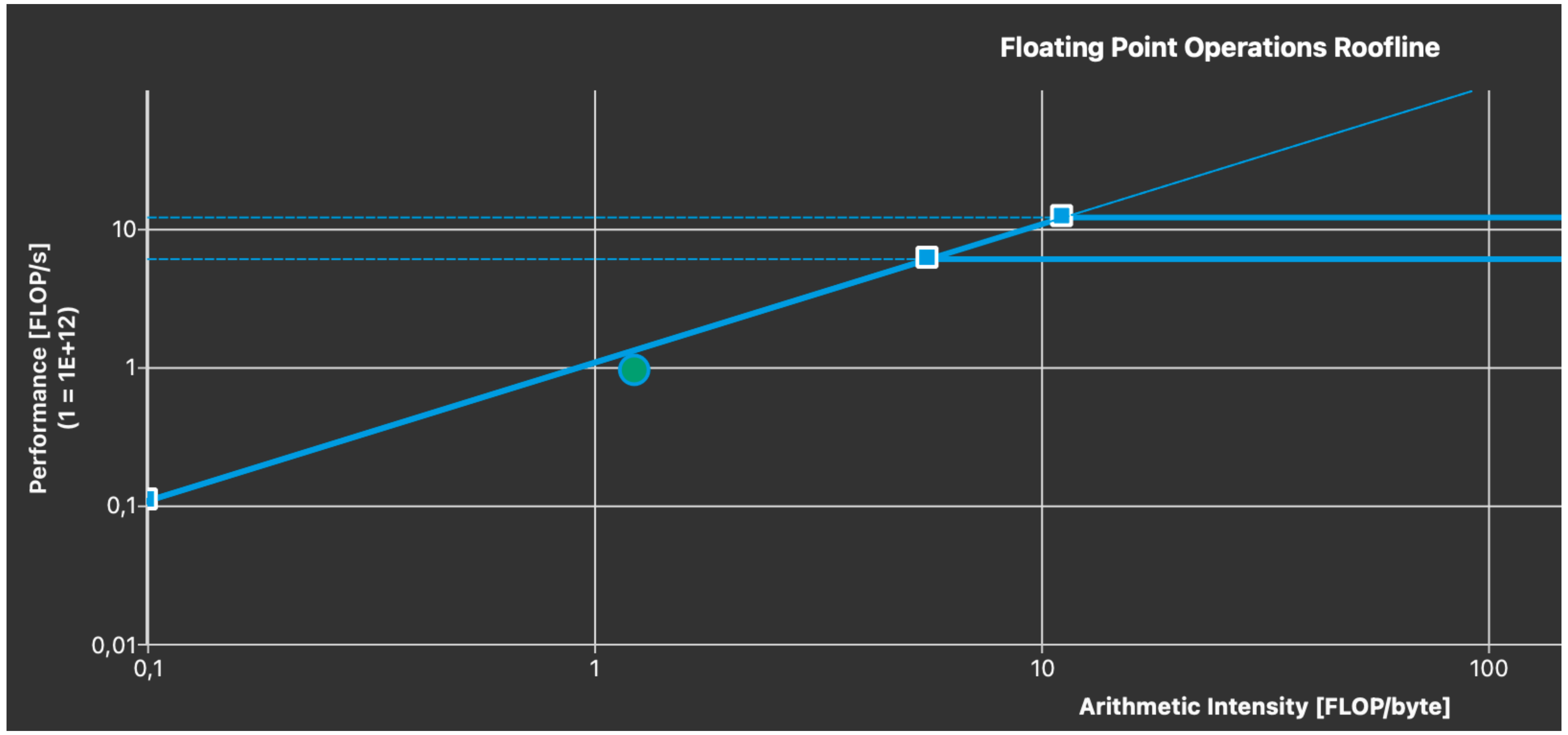
Profile GPU performance with NSight

Demo

Example: Jacobi method

Profile GPU performance with NSight





→ memory-bound, *not* compute-bound

other findings:

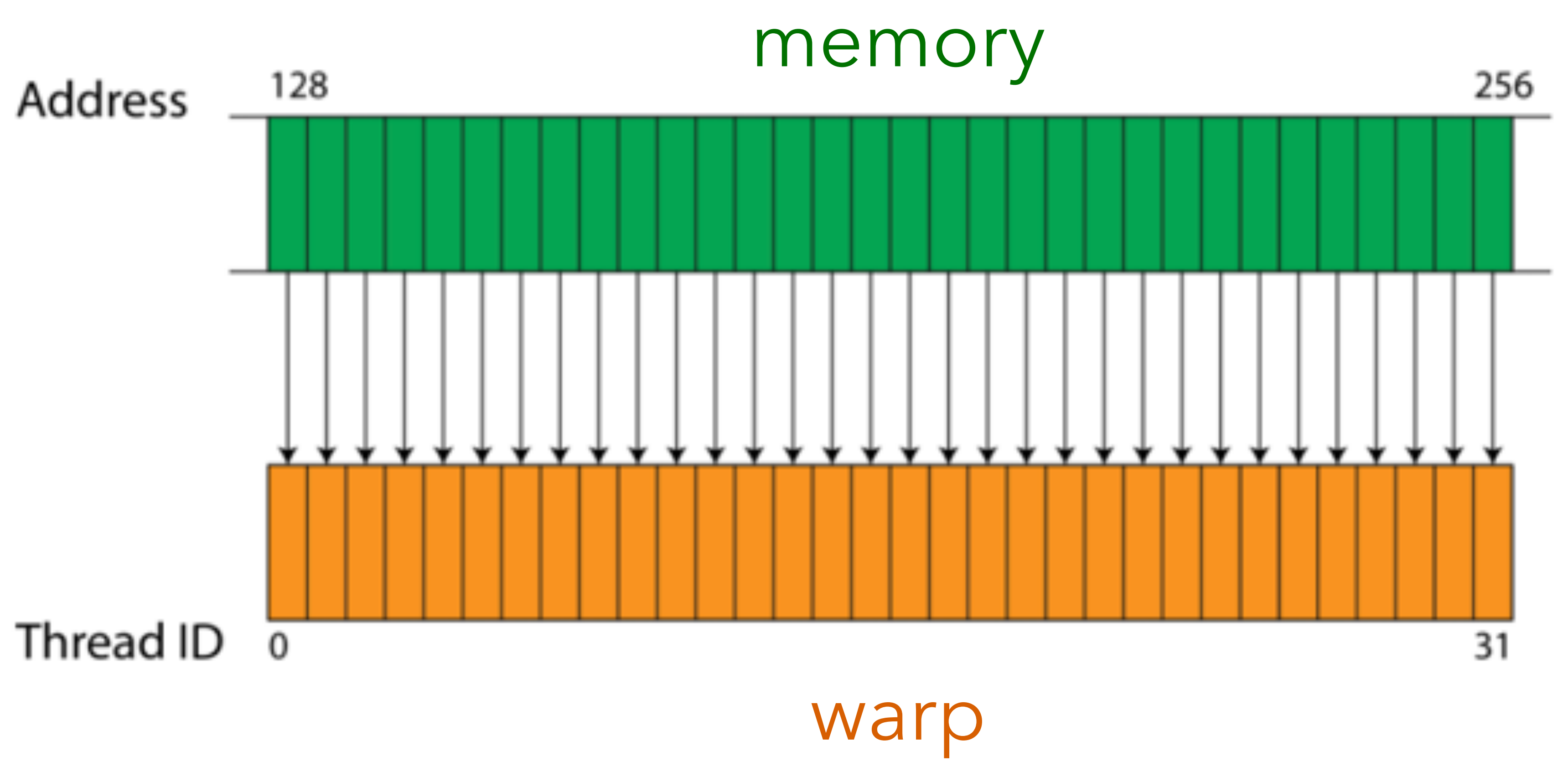
- 10 % uncoalesced global memory accesses (uncoal-a-what?)
- branch efficency: 0 (this can't be good, right?)

Advanced Topics

Advanced Topics

Coalesced memory access

- Uncached global memory latency: ~400 cycles
  - L2/L1 reduces this to 200/20 cycles, but limited size
- Single-precision instruction: ~4 cycles → try to limit memory reads/writes!!!
- very important consideration: 32 threads (= 1 warp) can combine („coalesce“) their memory reads/writes into a single transaction, speed-ups by up to 32x possible
- use structure of arrays instead array of structures



```
data[i + thread_id] = …
```
✓

```
data[i] = …
```
✗

```
                  j ≠ 1
data[i + j * thread_id] = …
```
✗

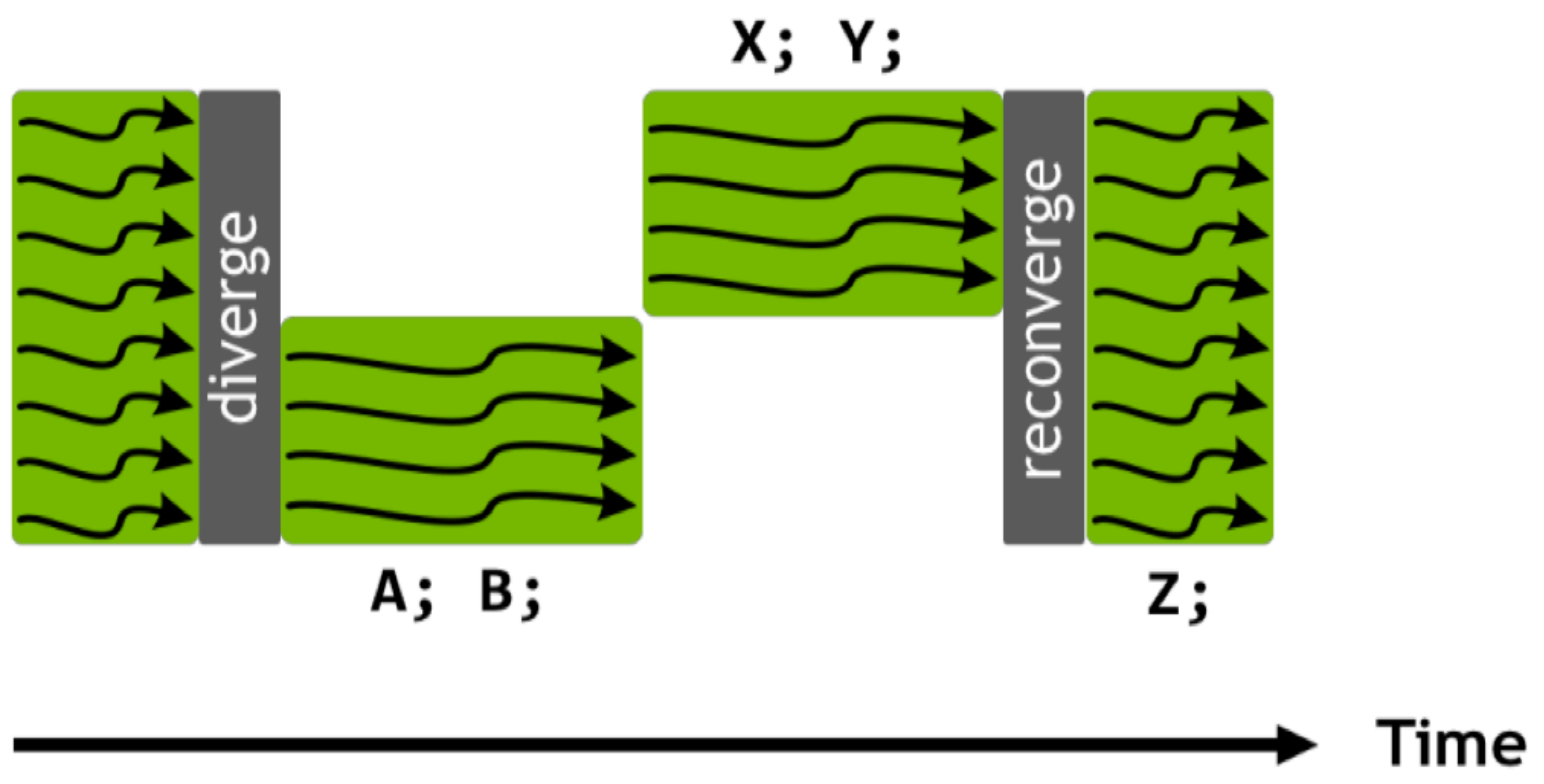
Advanced Topics

Branch divergence

- warp (= 32 threads) operates in lock-step (Single Instruction Multiple Threads)
- if only some threads fulfill an if-condition, all others are idle while the code within the if-block is run
- (it's more complicated+flexible on latest hardware, e.g. V100, A100, but performance still reduced)

➡ try to ensure that if-conditions evaluate the same for all threads in a given warp

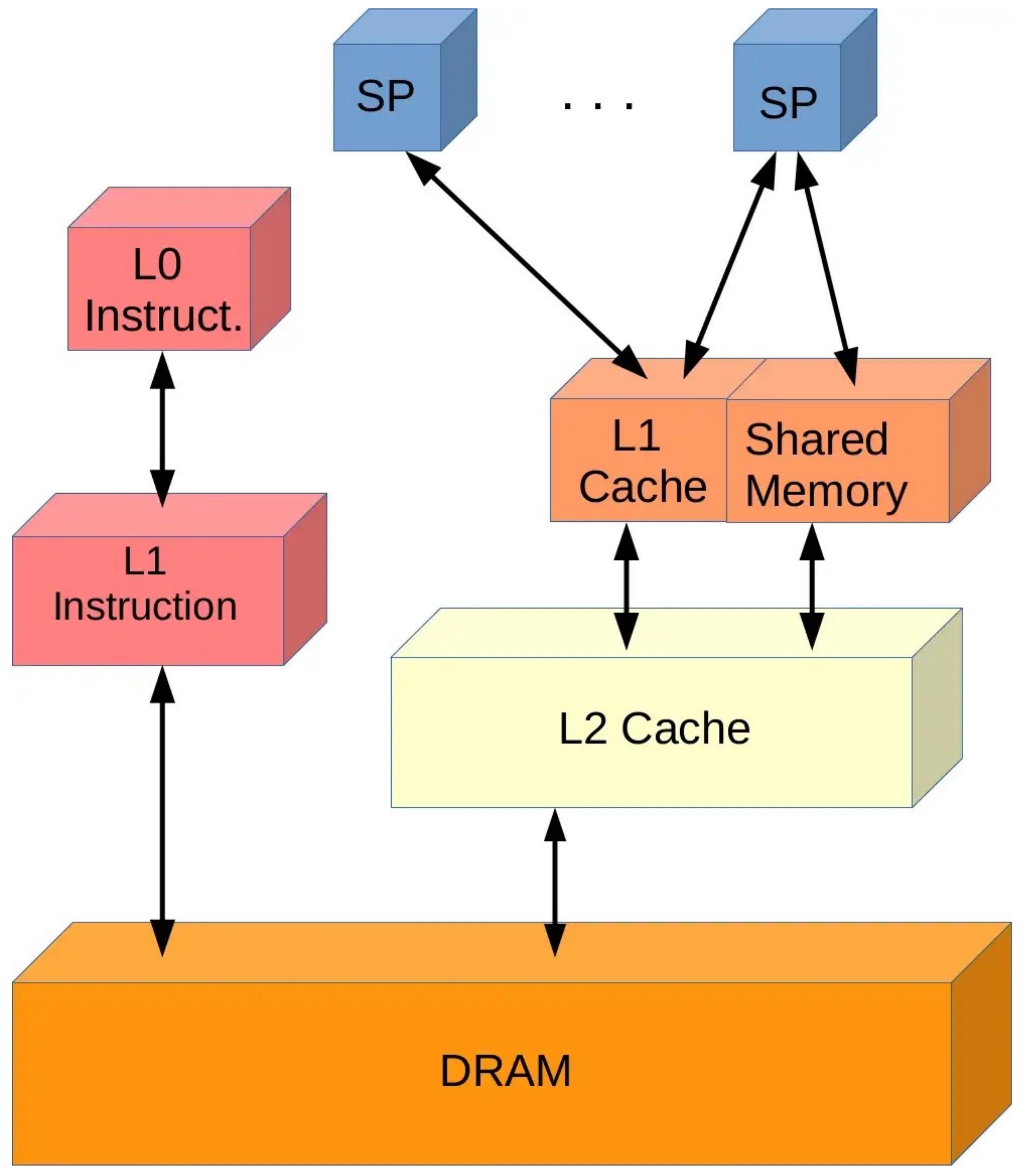
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

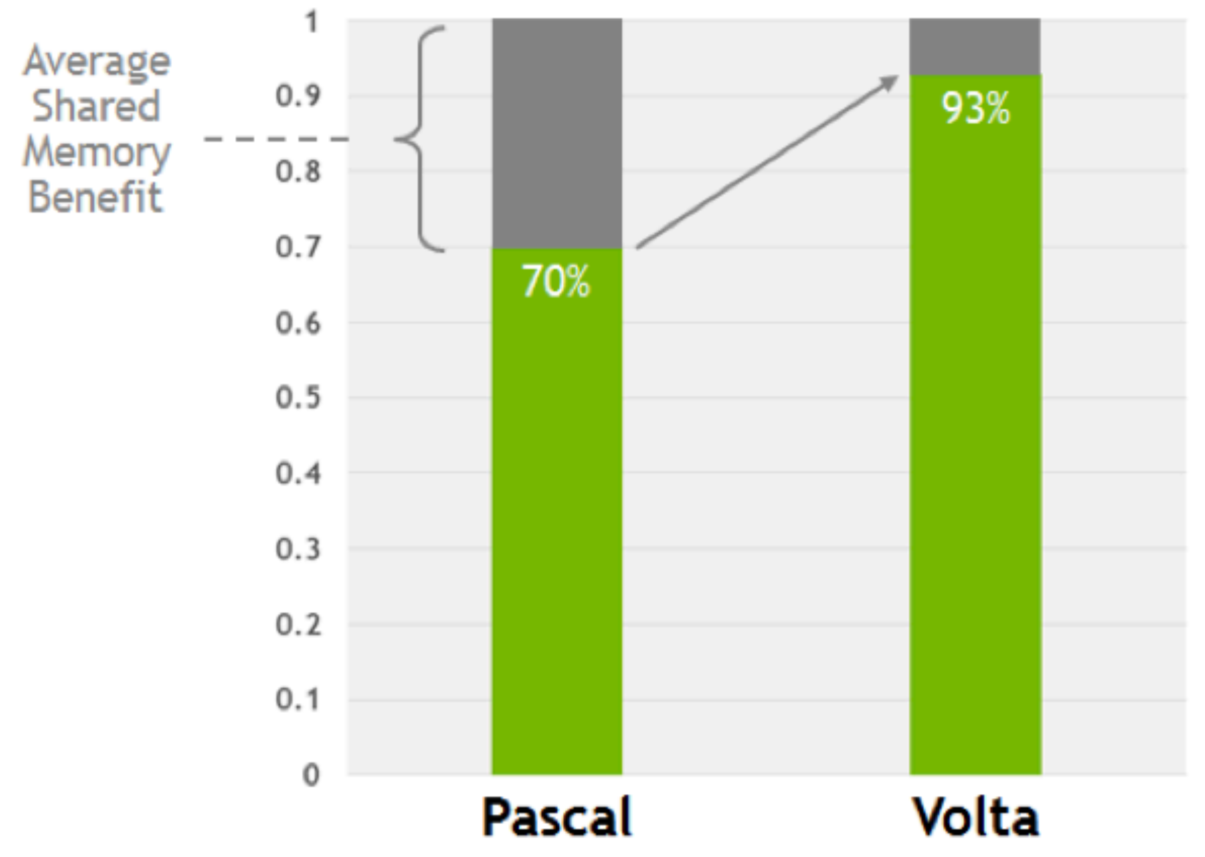


Advanced Topics

Shared Memory

- Memory that is shared between all threads in a block
- Uncached global memory latency: ~400 cycles
  - L2/L1 reduces this to 200/20 cycles, but limited size
- Shared Memory ~20 cycles (but also limited size, in fact L1 = shared memory on new GPU)
- Used to be very important, because one can not directly control what is kept by L1
- With modern data-center NVidia GPU (V100, A100), not as important, because L1 is quite smart as claimed by NVidia and independently confirmed [arXiv:1804.06826]
  - this is also my experience, also for this lecture's Jacobi example
  - perhaps still useful for complicated memory access patterns
  - … but of course automatic caching leads to much simpler code!





Conclusions

Conclusions

GPU for HEP

- Increasing number of GPU-aided new developments in HEP
- Calculations can be sped up by factors of $\mathcal{O}(10\ldots100)$ by putting parallelisable algorithms on the GPU
- Allows to study more complex systems (higher multi etc.)
- Theoretically also much better energy and cost efficiency
- Hardware increasingly available, software tooling easier to use

Tutorial

Tutorial

Port phase-space integral

- Consider cross section („Introduction to Event Generators", slide 47)

$$\sigma = \int f_i\left(x_1,\mu^2\right) f_j\left(x_2,\mu^2\right)\frac{1}{F}\bar{\sum}|M|^2\Theta(\text{cuts})\mathrm{d}x_1\ \mathrm{d}x_2\ \mathrm{d}\Phi_n$$

- Divide and conquer: begin with (massless) phase-space sampling

$$V_n = \int \mathrm{d}\Phi_n = \int \delta^4\left(P - \sum_{i=1}^{n} p_i\right)\prod_{i=1}^{n}\left(\mathrm{d}^4 p_i \delta\left(p_i^2\right)\theta\left(p_i^0\right)\right)$$

- Use flat sampling (Rambo algorithm), port from CPU → GPU

Backup

Tesla Model, Best Case Performance	**P4**	**P40**	**P100**	**P100**	**P100**	**V100**	**V100**	**V100**	**T4**	**A100**	**A100**
GPU	GP104	GP102	GP100	GP100	GP100	GV100	GV100	GV100	TU104	GA100	GA100
Bus	PCI-E 3.0	PCI-E 3.0	PCI-E 3.0	PCI-E 3.0	SXM	HGX-1	PCI-E 3.0	SXM2	PCI-E 3.0	PCI-E 4.0	SXM4
GDDR5 or GDDR6/HBM2 Memory	8 GB	24 GB	12 GB	16 GB	16 GB	16 GB	16/32 GB	16/32 GB	16 GB	40 GB	40 GB
Performance / Watt											
INT8 Efficiency (Gigaops/Watt)	290.7	188.0	-	-	-	334.9	224.0	209.3	1,857.1	6,240.0	6,240.0
FP16 TC, FP32 ACC Efficiency (Gigaflops/Watt)	-	-	-	-	-	666.7	448.0	416.7	930.4	1,560.0	1,560.0
FP16 Efficiency (Gigaflops/Watt)	-	-	74.8	74.8	70.7	141.3	100.5	104.7	-	195.0	195.0
FP32/TF32 Efficiency (Gigaflops/Watt)	72.7	47.2	37.2	37.2	35.3	70.7	50.2	52.3	116.3	1,040.0	1,040.0
FP64 Efficiency (Gigaflops/Watt)	2.3	1.5	18.8	18.8	17.7	35.3	25.0	26.0	3.6	48.8	48.8
$ / Performance											
Street Price, Single Unit	**$2,450**	***$5,000***	***$3,000***	***$3,500***	**$5,000**	***$5,000***	***$6,000***	**$7,500**	***$3,999***	***$8,500***	**$10,000**
$ / INT8 Gigaops	$112	$106	-	-	-	$100	$107	$119	$31	$7	$8
$ / FP16 TC, FP32 ACC Gigaflops	-	-	-	-	-	$50	$54	$60	$61	$14	$16
$ / FP16 Gigaflops	-	-	$160	$187	$236	$199	$214	$239	-	$109	$128
$ / FP32/TF32 Gigaflops	$450	$424	$323	$376	$472	$472	$478	$478	$491	$27	$32
$ / FP64 Gigaflops	$14,412	$13,514	$638	$745	$943	$943	$962	$962	$15,719	$436	$513
$ / Performance / Watt											
$ / INT8 Gigaops / Watt	$8.43	$26.60	-	-	-	$14.93	$26.79	$35.83	$2.15	$1.36	$1.60
$ / FP16 TC, FP32 ACC Gigaflops / Watt	-	-	-	-	-	$7.50	$13.39	$18.00	$4.30	$5.45	$6.41
$ / FP16 Gigaflops / Watt	-	-	$40.11	$46.79	$70.75	$35.38	$59.71	$71.66	-	$43.59	$51.28
$ / FP32/TF32 Gigaflops / Watt	$33.72	$105.93	$80.65	$94.09	$141.51	$70.75	$119.43	$143.31	$34.39	$8.17	$9.62
$ / FP64 Gigaflops / Watt	$1,080.88	$3,378.38	$159.57	$186.17	$283.02	$141.51	$240.38	$288.46	$1,100.35	$174.36	$205.13
* *Base Teraops or Teraflops unknown*											