QCD Lunch, CERN

25 February 2020

A Gentle Introduction to GPU programming for TH

Stephen Jones

Amdahl's Law

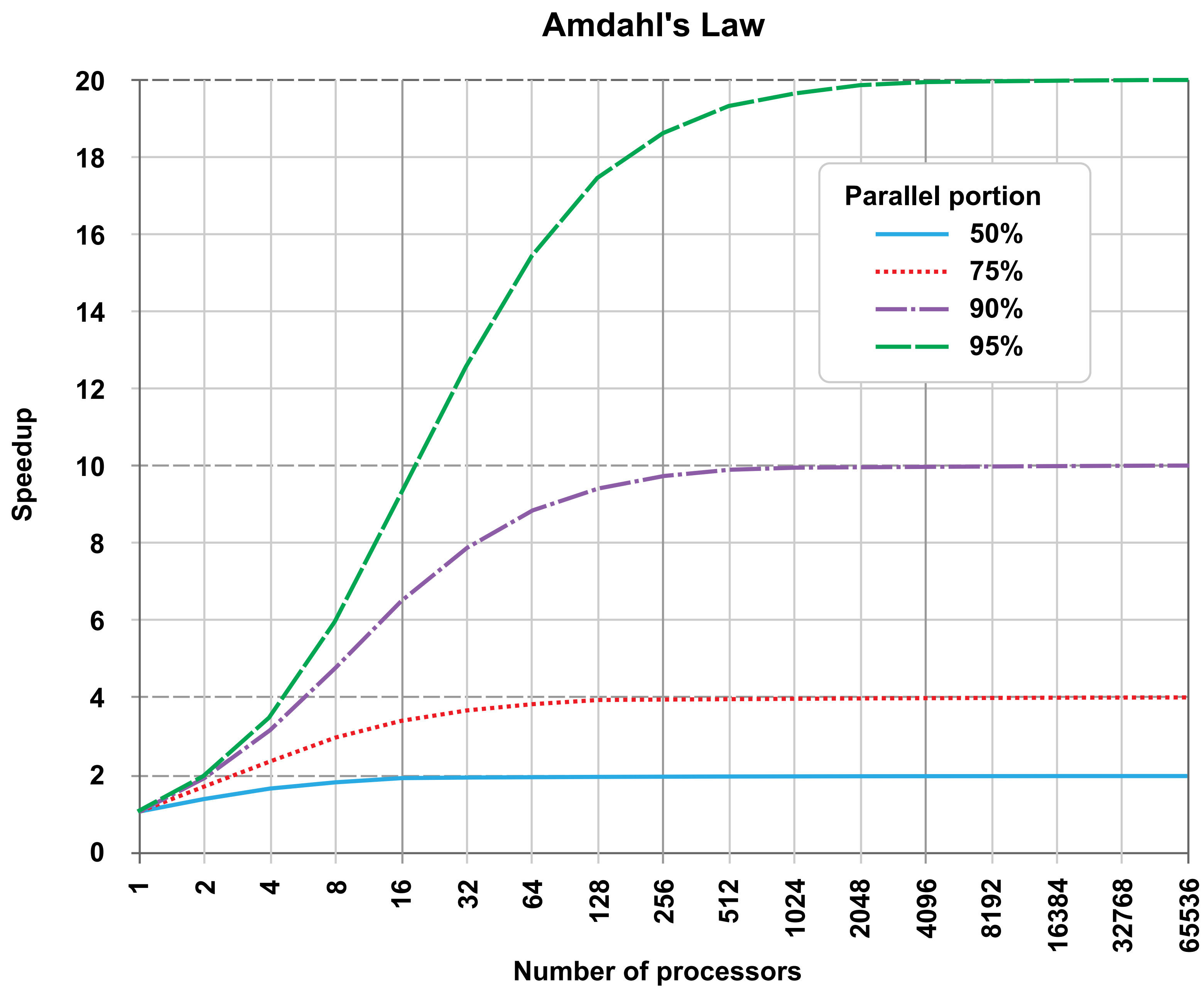
First decide if/where to speedup your code

$$S(x) = \frac{1}{(1-p) + \frac{p}{x}}$$

S - whole task speedup

x - speedup of part of task

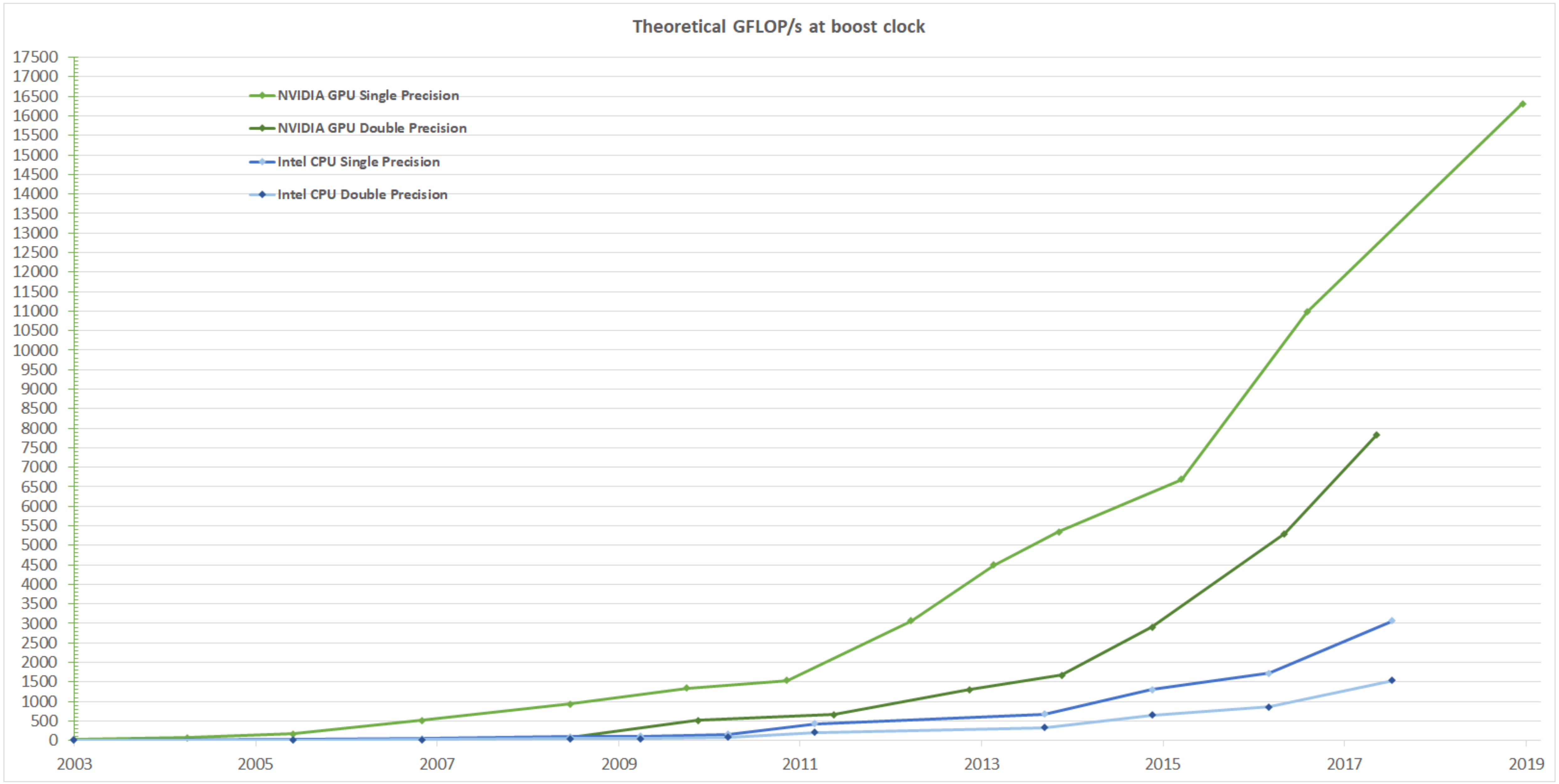
p - initial proportion of execution time of part of task



Wikipedia, CC BY-SA 3.0

Why bother with GPUs?

(Currently) GPUs typically have higher FLOP performance and memory bandwidth than CPUs (total / per $ / per J)



Nvidia

GPU Programming: Pick your poison

OpenACC

OpenMP style pragmas

Dev: Cray/CAPS/Nvidia/PGI

```
void saxpy(int n, float a, float * restrict
x, float * restrict y)
{
#pragma acc kernels
  for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
```

OpenCL

Based on C/C++ (since OpenCL 2.2)

Dev: Apple (deprecated), Khronos

```
const char *saxpy_kernel =
"__kernel                                   \n"
"void saxpy_kernel(float alpha,             \n"
"                    __global float *A      \n"
"                    __global float *B      \n"
"                    __global float *C)     \n"
"{                                          \n"
"   // Get the index of the work item       \n"
"   int index = get_global_id(0);           \n"
"   C[index] = alpha * A[index] + B[index]; \n"
"}                                          \n";
```

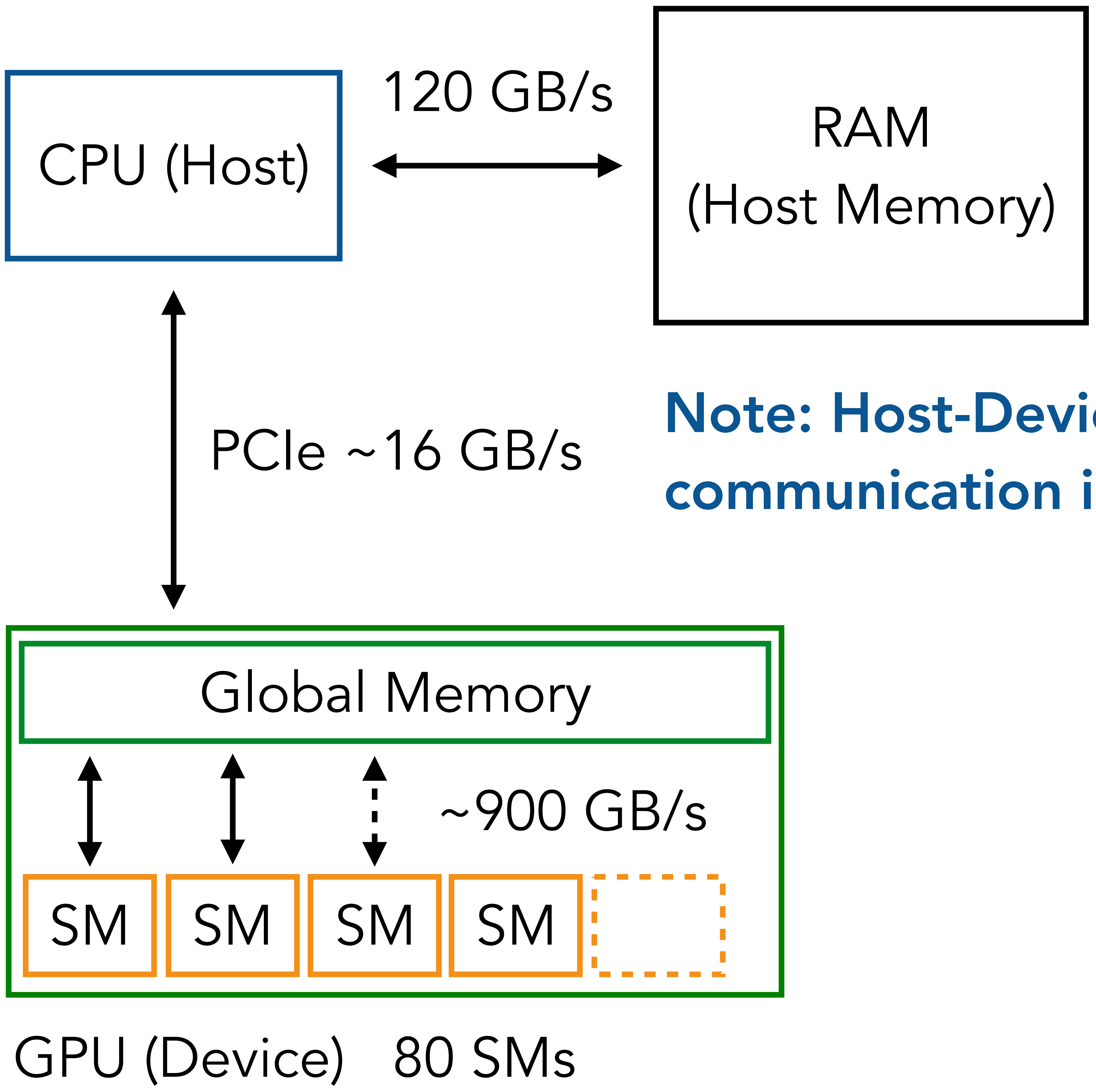
CUDA

Based on C/C++

Dev: Nvidia

← **this talk**

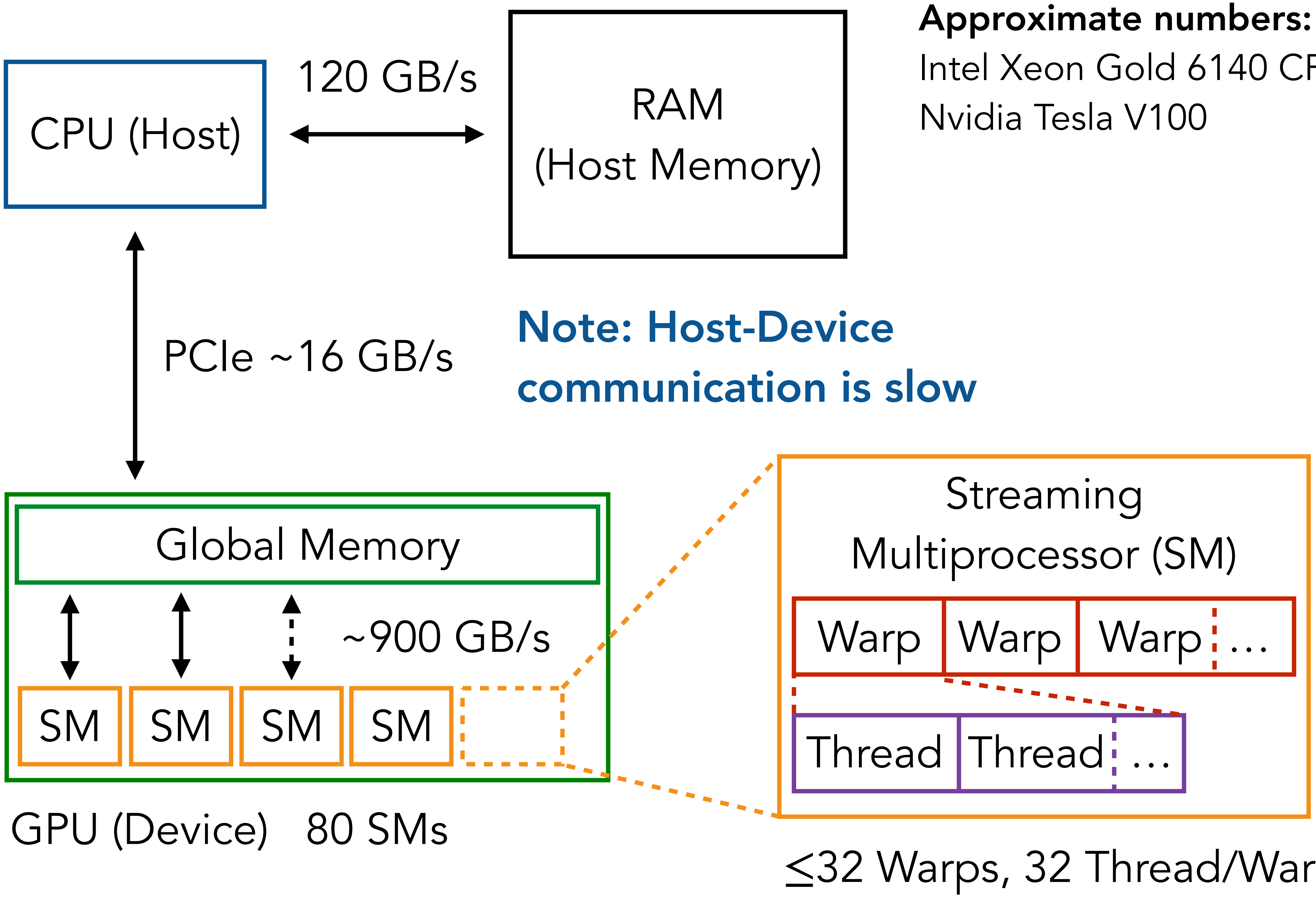
Host Device Layout



Approximate numbers:
Intel Xeon Gold 6140 CPU
Nvidia Tesla V100

Note: Host-Device communication is slow

Host Device Layout



Approximate numbers:
Intel Xeon Gold 6140 CPU
Nvidia Tesla V100

Note: Host-Device communication is slow

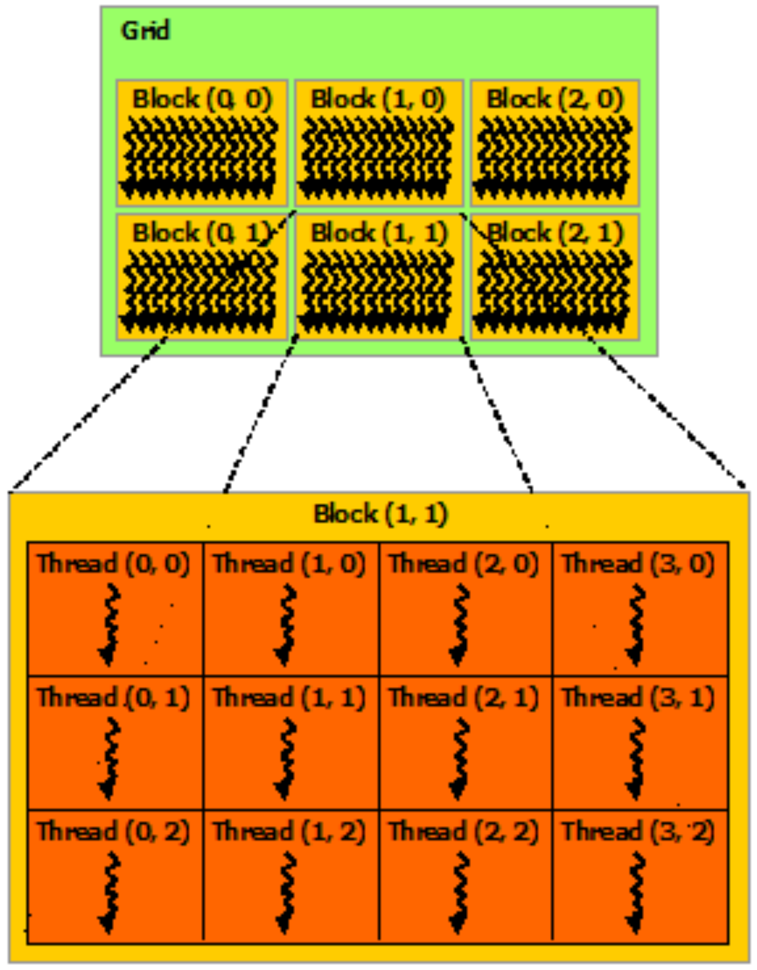
Grid/Block/Thread Layout

We will write and invoke **kernels**:
`my_kernel<<<num_blocks,num_threads_per_block>>>();`

kernel: consists of a single **grid**
grid: contains several **blocks**, controlled by `num_blocks`
block: contains several **threads**, controlled by `num_threads_per_block`

The threads in a thread block are grouped into **warps**, all threads in a warp execute the same instruction* (but on different data) in parallel:
Single Instruction Multiple Threads (SIMT)

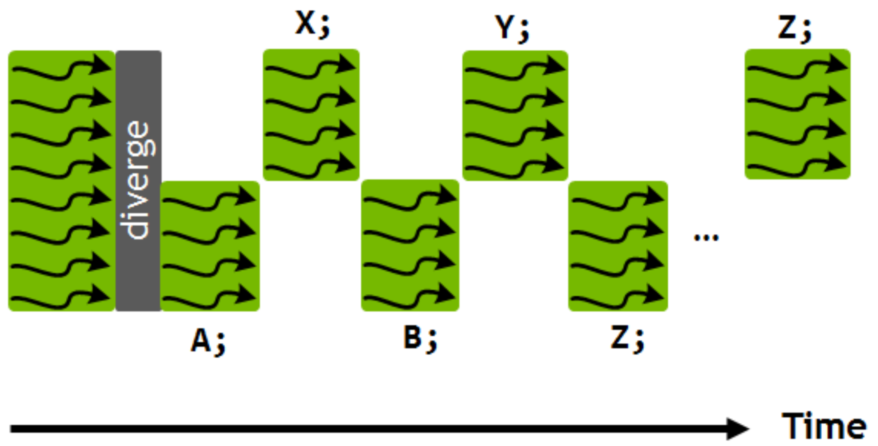
Blocks are distributed to SMs with available execution capacity.
The threads of a thread block execute concurrently on one SM, and multiple thread blocks can execute concurrently on one multiprocessor.
As thread blocks terminate, new blocks are launched on the vacated multiprocessors.



Nvidia

Warp divergence*

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



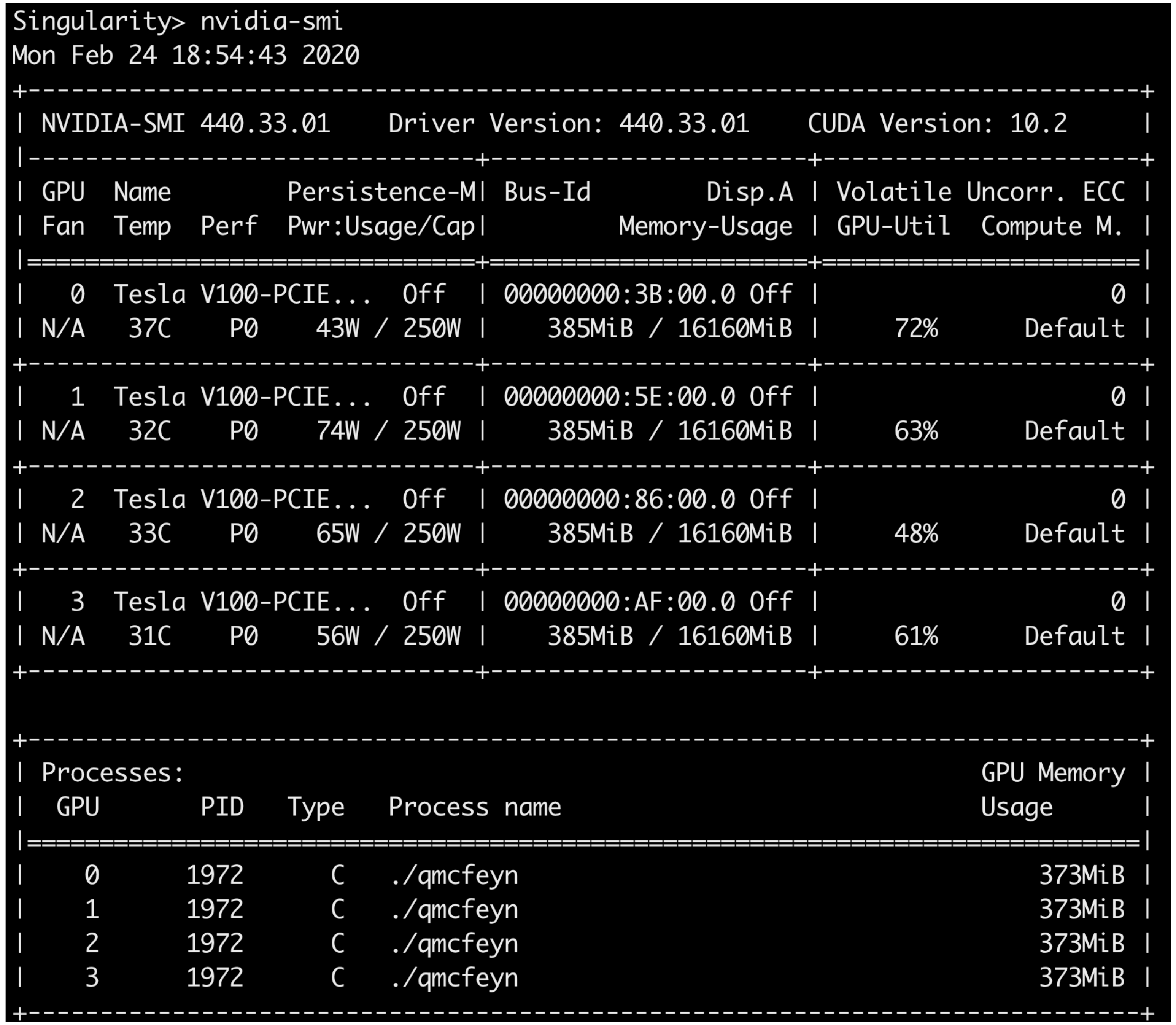
Nvidia

	Compute Capability							
Technical Specifications	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.5**
Maximum number of resident grids per device	32		16	128	32	16	128	
Maximum dimensionality of grid of thread blocks	3							
Maximum x-dimension of a grid of thread blocks ●	2^{31}-1							
Maximum y- or z-dimension of a grid of thread blocks	65535							
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block ●	1024							
Maximum z-dimension of a block	64							
Maximum number of threads per block ●	1024							
Warp size ●	32							
Maximum number of resident blocks per multiprocessor	32							16
Maximum number of resident warps per multiprocessor ●	64							32
Maximum number of resident threads per multiprocessor	2048							1024
Number of 32-bit registers per multiprocessor	64 K							
Maximum number of 32-bit registers per thread block	64 K		32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	255							
Maximum amount of shared memory per multiprocessor	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB	64 KB
Maximum amount of shared memory per thread block	48 KB						96 KB	64 KB
Number of shared memory banks	32							
Amount of local memory per thread	512 KB							
Constant memory size	64 KB							
Cache working set per multiprocessor for constant memory	8 KB			4 KB	8 KB			
Maximum number of instructions per kernel ●	512 million							

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

nvidia-smi

htop like program for monitoring GPU activity



```
Singularity> nvidia-smi
Mon Feb 24 18:54:43 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.33.01    Driver Version: 440.33.01    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-PCIE...  Off  | 00000000:3B:00.0 Off |                    0 |
| N/A   37C    P0    43W / 250W |    385MiB / 16160MiB |     72%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-PCIE...  Off  | 00000000:5E:00.0 Off |                    0 |
| N/A   32C    P0    74W / 250W |    385MiB / 16160MiB |     63%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-PCIE...  Off  | 00000000:86:00.0 Off |                    0 |
| N/A   33C    P0    65W / 250W |    385MiB / 16160MiB |     48%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-PCIE...  Off  | 00000000:AF:00.0 Off |                    0 |
| N/A   31C    P0    56W / 250W |    385MiB / 16160MiB |     61%      Default |
+-------------------------------+----------------------+----------------------+


+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1972      C   ./qmcfeyn                                   373MiB |
|    1      1972      C   ./qmcfeyn                                   373MiB |
|    2      1972      C   ./qmcfeyn                                   373MiB |
|    3      1972      C   ./qmcfeyn                                   373MiB |
+-----------------------------------------------------------------------------+
```

Code Examples: gpu-tut

Quasi-Monte Carlo (Rank 1 Lattices)

Quasi-Monte Carlo (QMC) in a Weighted Function Space

First applications to loop integrals, see: Li, Wang, Yan, Zhao 15; de Doncker, Almulihi, Yuasa 17, 18; de Doncker, Almulihi 17; Kato, de Doncker, Ishikawa, Yuasa 18

$$I[f] \approx \bar{Q}_{n,m}[f] \equiv \frac{1}{m}\sum_{k=0}^{m-1} Q_n^{(k)}[f], \qquad Q_n^{(k)}[f] \equiv \frac{1}{n}\sum_{i=0}^{n-1} f\left(\left\{\frac{i\mathbf{z}}{n} + \mathbf{\Delta}_k\right\}\right)$$

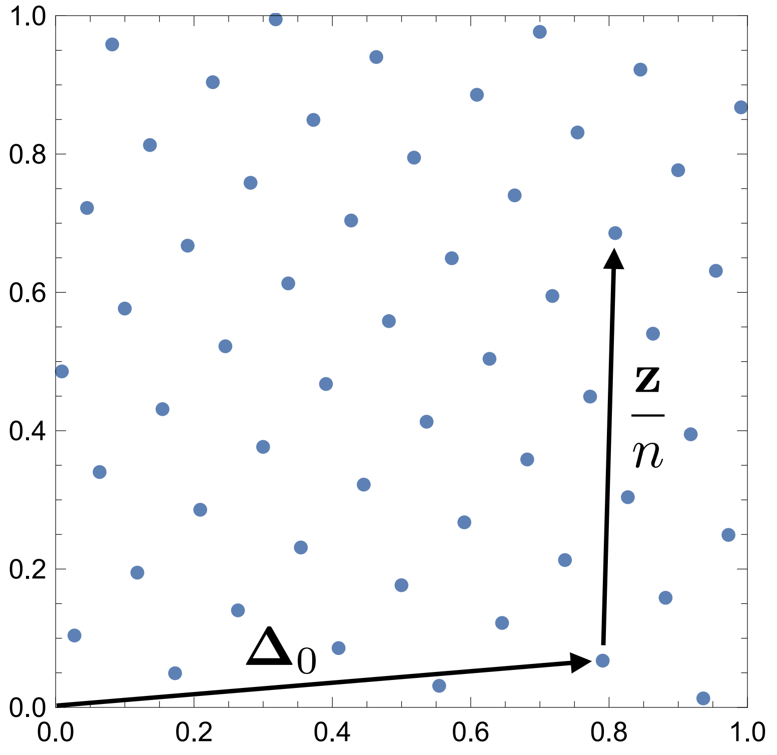
$\mathbf{z}$ - Generating vec.

$\mathbf{\Delta}_k$ - Random shift vec.

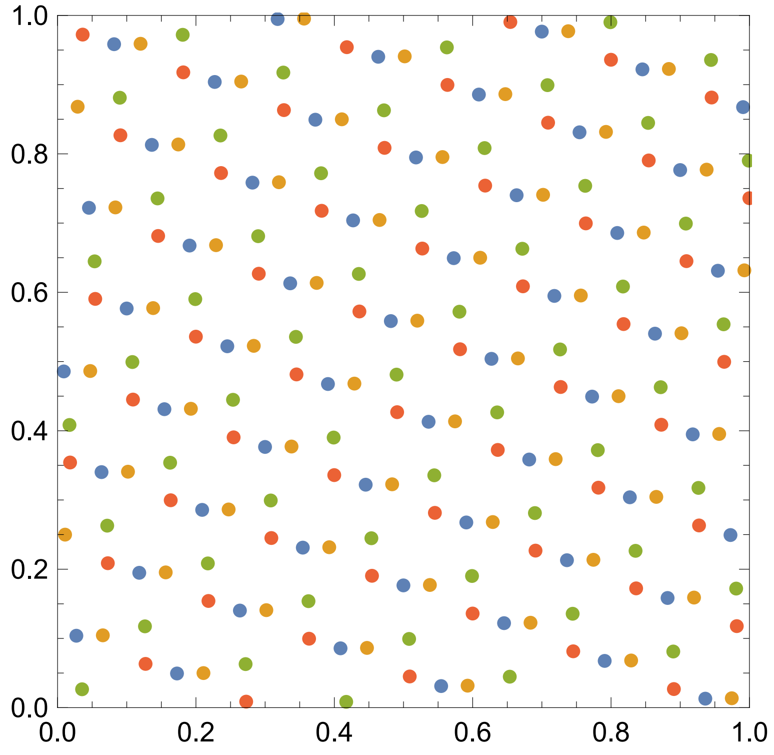
$\{\}$ - Fractional part

n - # Lattice points

m - # Random shifts



1 Shift

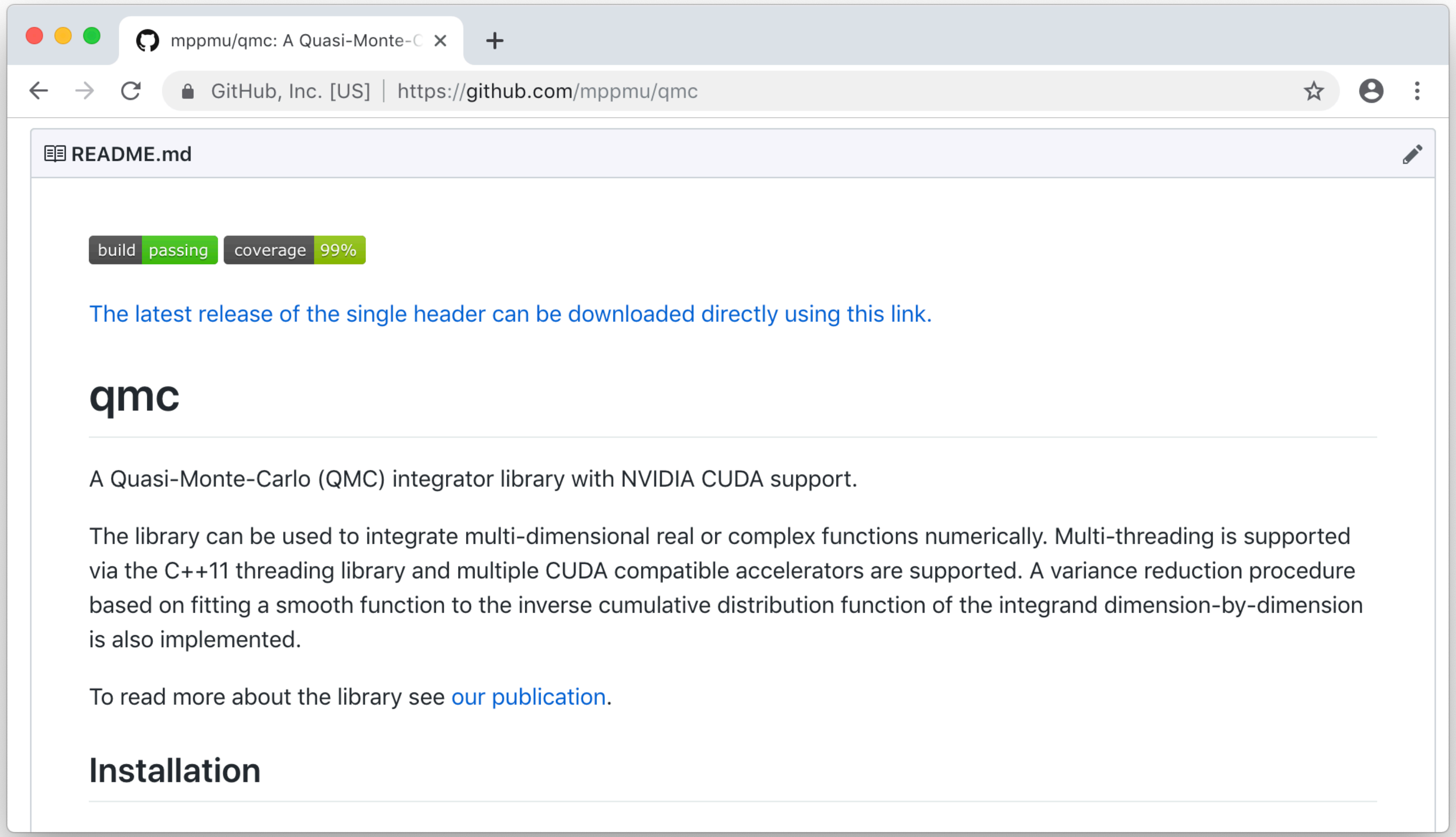


4 Shifts

Unbiased error estimate computed using (10-50) random shifts

qmc: Installation

qmc: distributed as a standalone single header c++11 library
quite flexible: define your own lattices, integral transforms, floating-point types (e.g… Boost multi-precision)


mppmu/qmc: A Quasi-Monte-C
GitHub, Inc. [US] | https://github.com/mppmu/qmc
README.md
build passing
coverage 99%

The latest release of the single header can be downloaded directly using this link.

qmc

A Quasi-Monte-Carlo (QMC) integrator library with NVIDIA CUDA support.

The library can be used to integrate multi-dimensional real or complex functions numerically. Multi-threading is supported via the C++11 threading library and multiple CUDA compatible accelerators are supported. A variance reduction procedure based on fitting a smooth function to the inverse cumulative distribution function of the integrand dimension-by-dimension is also implemented.

To read more about the library see our publication.

Installation

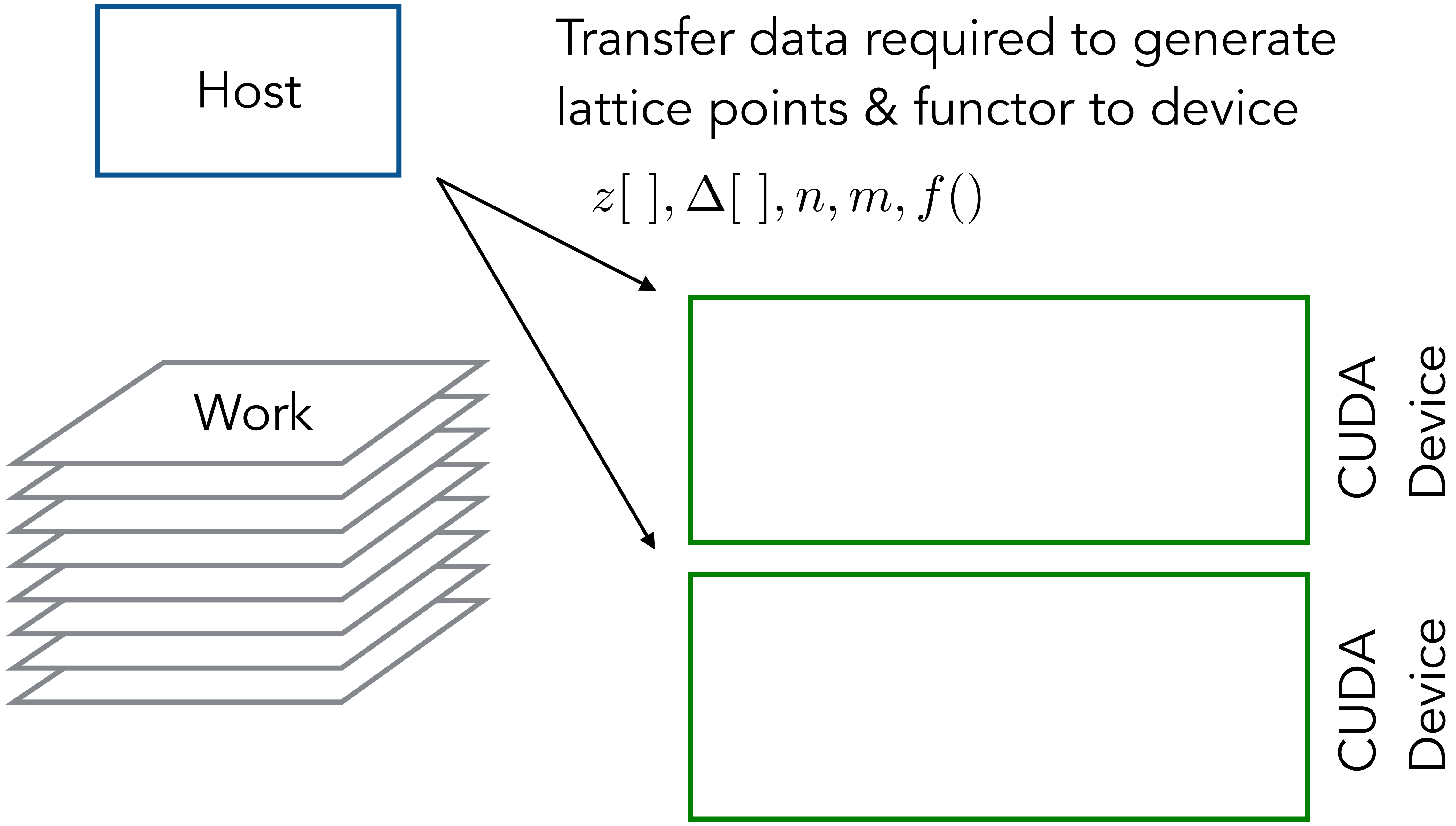
Publicly available (Github)

Extensive tests (CI) and coverage

Borowka, Heinrich, Jahn, SPJ, Kerner, Schlenk 18

qmc: Implementation

1) Devices are initialised

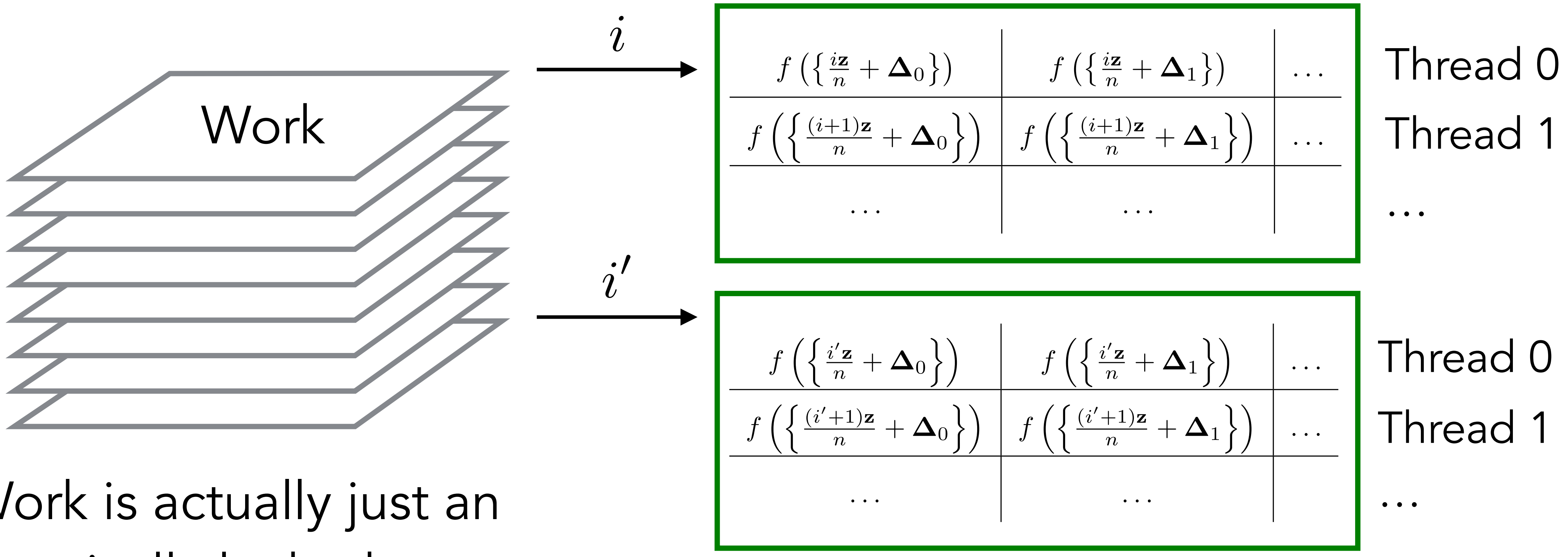

Host
Transfer data required to generate lattice points & functor to device
$z[\,], \Delta[\,], n, m, f()$
Work
CUDA Device
CUDA Device

qmc: Implementation

2) (Loop) Work is requested from central work queue by each device

Host

Only index of point(s) to calculate needs to be transferred
Partial results are accumulated by each thread on each device



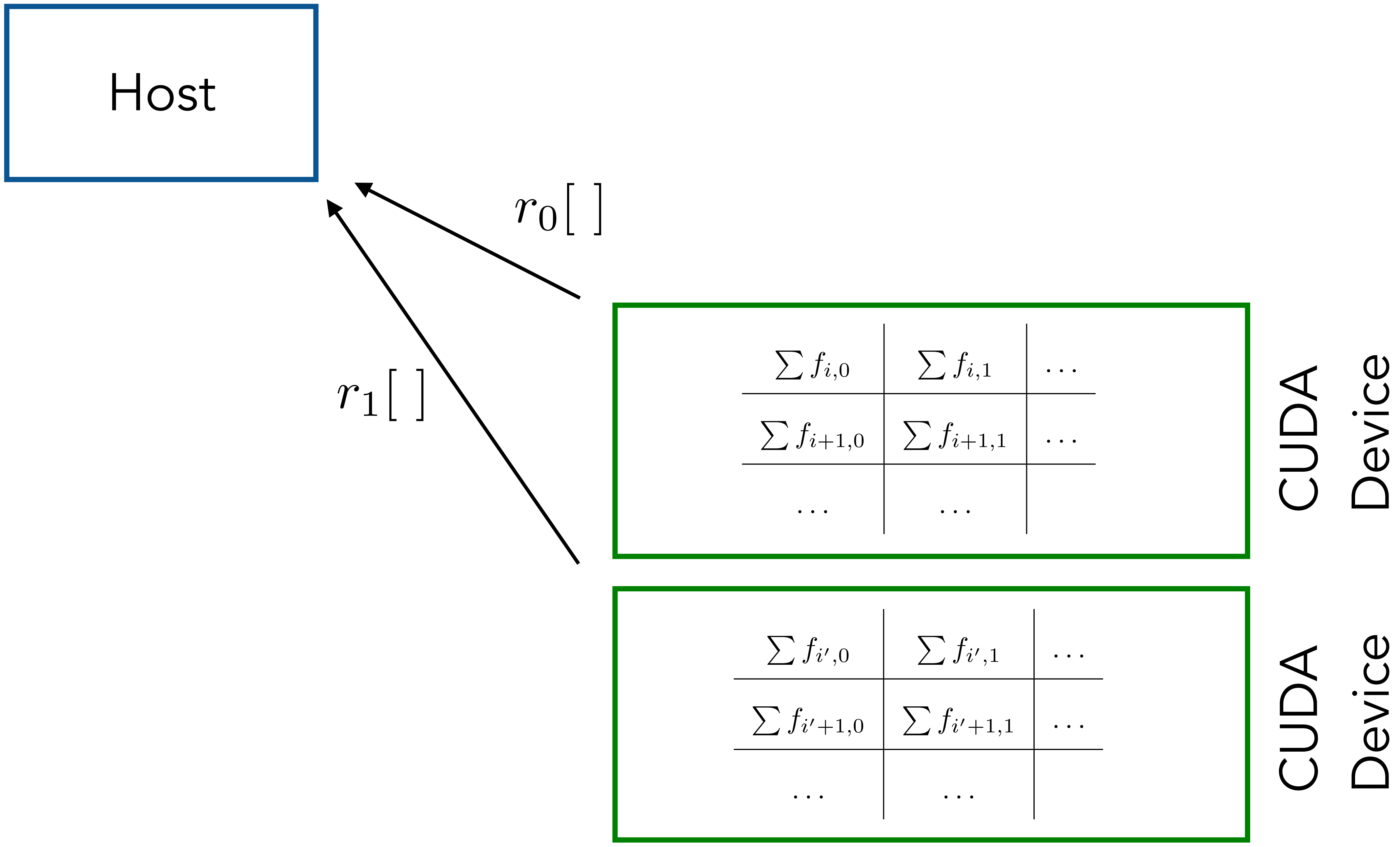
$f\left(\left\{\frac{i\mathbf{z}}{n}+\mathbf{\Delta}_0\right\}\right)$	$f\left(\left\{\frac{i\mathbf{z}}{n}+\mathbf{\Delta}_1\right\}\right)$	$\dots$
$f\left(\left\{\frac{(i+1)\mathbf{z}}{n}+\mathbf{\Delta}_0\right\}\right)$	$f\left(\left\{\frac{(i+1)\mathbf{z}}{n}+\mathbf{\Delta}_1\right\}\right)$	$\dots$
$\dots$	$\dots$	

$f\left(\left\{\frac{i'\mathbf{z}}{n}+\mathbf{\Delta}_0\right\}\right)$	$f\left(\left\{\frac{i'\mathbf{z}}{n}+\mathbf{\Delta}_1\right\}\right)$	$\dots$
$f\left(\left\{\frac{(i'+1)\mathbf{z}}{n}+\mathbf{\Delta}_0\right\}\right)$	$f\left(\left\{\frac{(i'+1)\mathbf{z}}{n}+\mathbf{\Delta}_1\right\}\right)$	$\dots$
$\dots$	$\dots$	

Work is actually just an atomically locked counter

qmc: Implementation

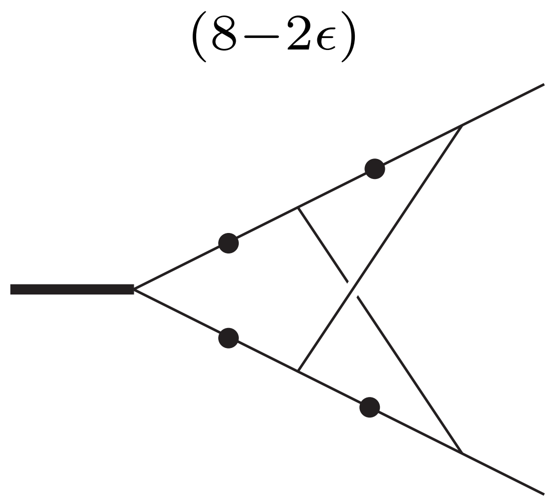
3) Once all work is completed, partial results are transferred back to host for final reduction



qmc: Usage

Let's compute a finite Feynman integral (à la de Doncker, Almulihi, Yuasa 17)

```
#include <iostream>
#include "qmc.hpp"
struct formfactor2L_t {
    const unsigned long long int number_of_integration_variables = 5;
#ifdef __CUDACC__
    __host__ __device__
#endif
    double operator()(const double arg[]) const
    {
        // Simplex to cube transformation
        double x0  = arg[0];
        double x1  = (1.-x0)*arg[1];
        double x2  = (1.-x0-x1)*arg[2];
        double x3  = (1.-x0-x1-x2)*arg[3];
        double x4  = (1.-x0-x1-x2-x3)*arg[4];
        double x5  = (1.-x0-x1-x2-x3-x4);
        double wgt =
        (1.-x0)*
        (1.-x0-x1)*
        (1.-x0-x1-x2)*
        (1.-x0-x1-x2-x3);
        if(wgt <= 0) return 0;
        // Integrand
        double u=x2*(x3+x4)+x1*(x2+x3+x4)+(x2+x3+x4)*x5+x0*(x1+x3+x4+x5);
        double f=x1*x2*x4+x0*x2*(x1+x3+x4)+x0*(x2+x3)*x5;
        double n=x0*x1*x2*x3;
        double d = f*f*u*u;
        return wgt*n/d;
    }
} formfactor2L;
```



Note: can compile this code with or without CUDA

```
int main() {
    integrators::Qmc<double,double,5,integrators::transforms::Korobov<3>::type> integrator;
    integrator.minn = 100000000; // (optional) lattice size
    integrators::result<double> result = integrator.integrate(formfactor2L);
    std::cout << "integral = " << result.integral << std::endl;
    std::cout << "error    = " << result.error    << std::endl;
    return 0;
}
```

```
$ nvcc -O3 -arch=sm_70 -std=c++11 -x cu -I../src 102_ff2_demo.cpp -o 102_ff2_demo.out -lgsl -lgslcblas && ./102_ff2_demo.out
integral = 0.27621
error    = 4.49751e-07
```

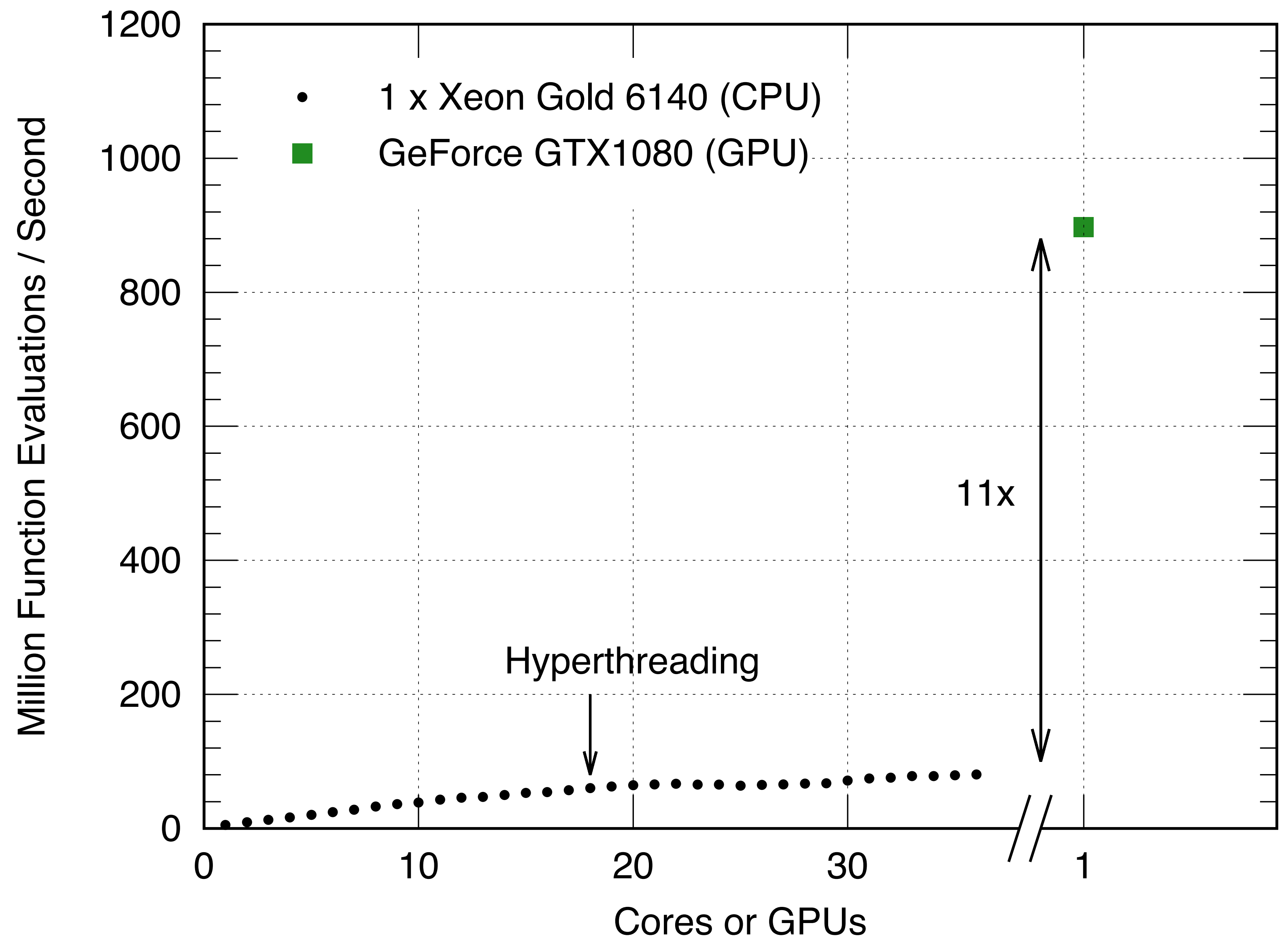
(Agrees with analytic result)

Code Examples: gpu-tut

qmc: Performance

Accuracy limited by number of function evaluations

Can accelerate this using Graphics Processing Units (GPUs)



Device	M Func. Evals/s	Speedup
Xeon 6140	80.6	-
GTX1080	897	11x

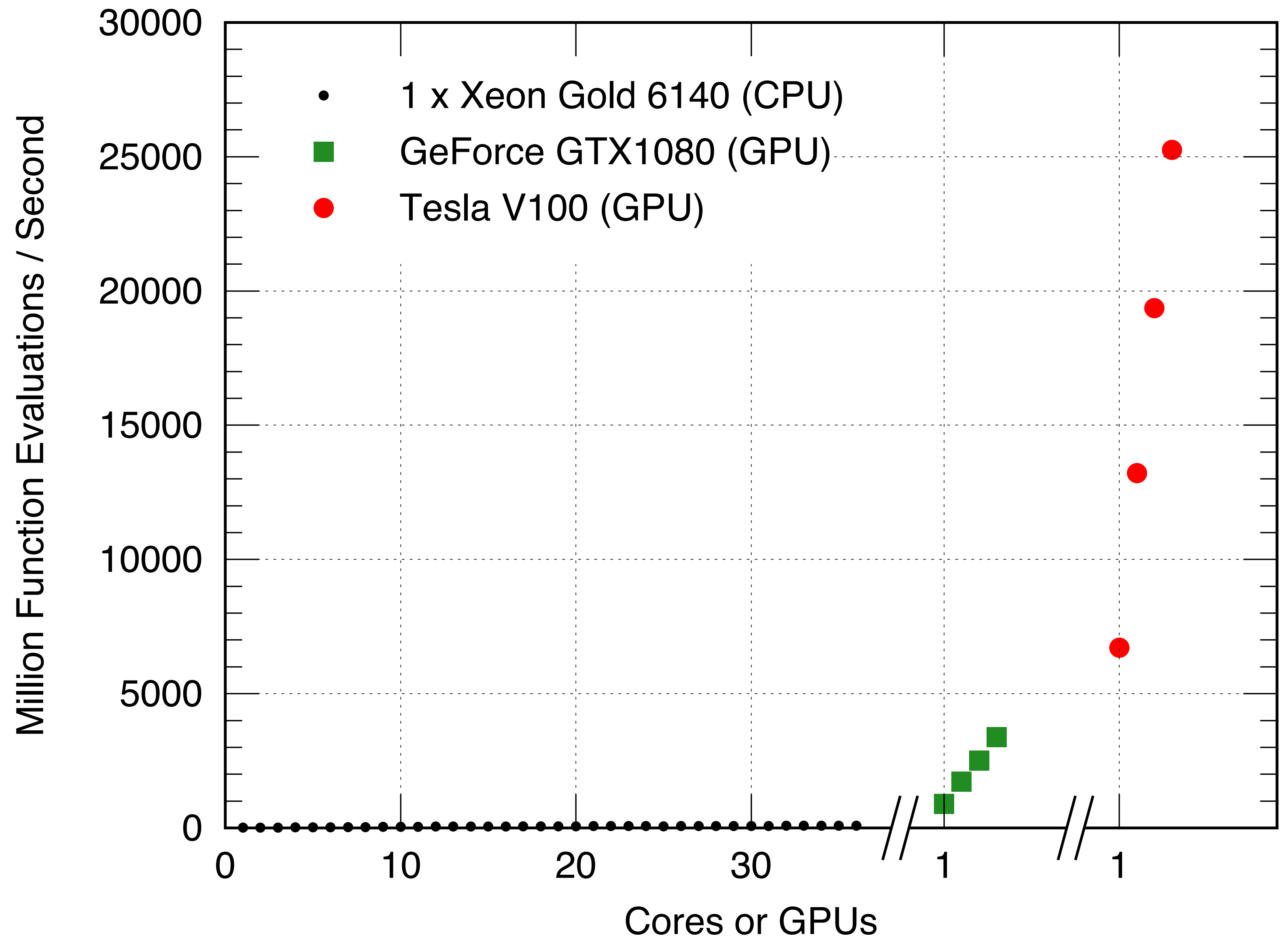
Note: Performance gain highly dependent on integrand & hardware!

Still room for further optimisations (both for CPU and GPU)

qmc: Performance

Accuracy limited by number of function evaluations

Can accelerate this using Graphics Processing Units (GPUs)



Device	M Func. Evals/s	Speedup
Xeon 6140	80.6	-
GTX1080	897	11x
Tesla V100	6710	83x

Note: Performance gain highly dependent on integrand & hardware!

Still room for further optimisations (both for CPU and GPU)

Happy Pancake Day

Thank you for listening!