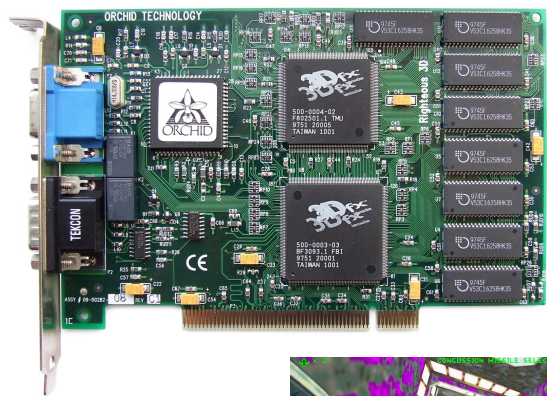


GPUs: The Good, The Bad, and the Ugly

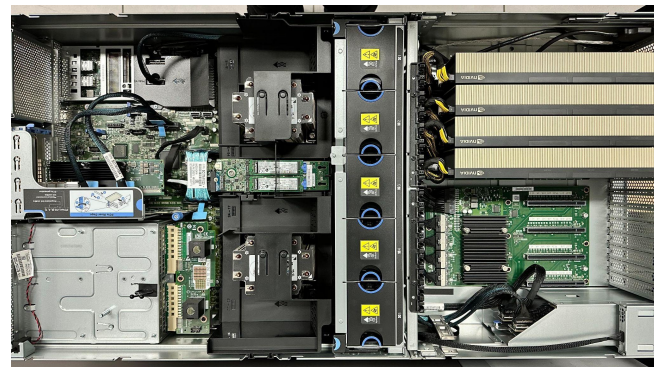
Robert Blackwell, G raud Krawezik
Scientific Computing Core

From video games to data centers?

1996



2022



Outline

- GPUs: past, present and future [Géraud]
- GPU programming and performance [Robert]
- The fine prints about GPU programming [Géraud]

What are GPUs?

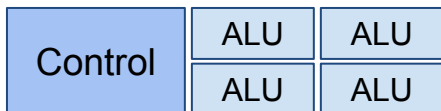
- Graphics Processing Unit
- Early 1990s: 3D accelerators
 - They were separated from the video card!
- These got merged, and what we call now GPUs are their offspring
- Since 2010, their performance with dense matrix operations have made them a key to HPC systems: “GPGPU”

Computing on GPUs: GPGPU

- Initially, graphics-specific languages: OpenGL, DirectX
 - **2003**: Krüger, Westermann “Linear algebra operators for GPU implementation of numerical algorithms”
 - **2003**: Bolz, Farmer, Grinspun, Schröder: “Sparse matrix solvers on the GPU: conjugate gradients and multigrid”
- **2007**: First CUDA version
- **2008**: HPC paper: Volkov + Demmel: “Benchmarking GPUs to Tune Dense Linear Algebra”
- **2009**: First ML paper: Raina, Madhavan, Ng: “Large-scale Deep Unsupervised Learning using Graphics Processors”

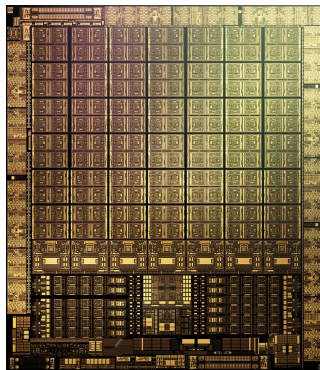
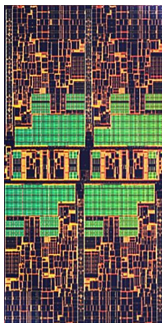
How do GPUs differ from CPUs?

CPU

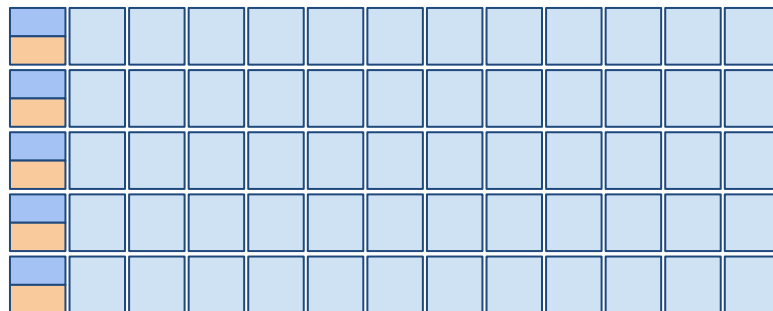


Cache

RAM



GPU



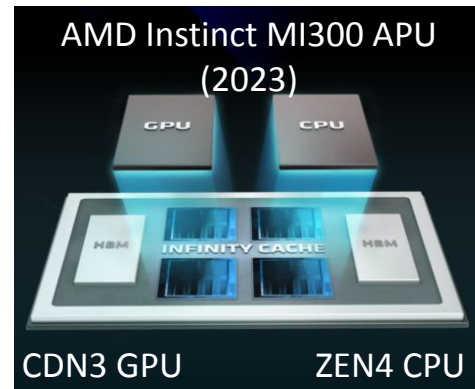
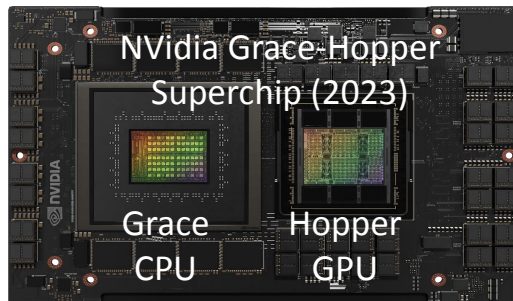
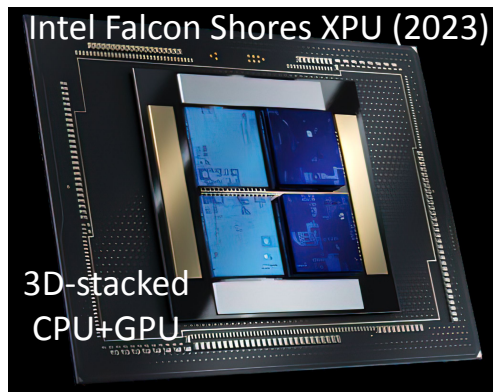
RAM

- General purpose
- < 100 cores “Multi core”
- Large cache
- ~ 11 billion transistors

- Special purpose
- > 1000 cores “Many core”
- Control logic is fairly limited
- Small programmable cache
- High bandwidth memory
- ~ 80 billion transistors

The Future: CPU + GPU integration

Looking at the vendors' roadmaps



- On the same socket:
 - CPU cores
 - GPU cores
 - HBM memory on some modules
- **Both CPU and GPU can address the same memory space**

The GPU market

- A very preeminent vendor has most of the market
 - BLAS closed source
 - FFT closed source
 - HPLinpack closed source
- However CPU companies are coming out with GPUs for HPC/ML:
 - AMD Instinct MI (used in Oak Ridge's Frontier)
 - Intel Ponte Vecchio (used in Argonne's Aurora)

GPU Programming is hard

- Bad or no optimizations often means sluggish code
- For non experts, it can feel overwhelming:
 - Global memory
 - Texture memory
 - Shared memory
 - Local memory
 - L1/L2/constant cache
 - Warps, blocks, grids
 - Tensor cores
 - Streams
 - Coalesced accesses, memory bank conflicts

But there are vendor optimized libraries

- BLAS
- FFT
- DNN
- Tensors
- Solvers
- PyTorch, TensorFlow integration

Performance and programming examples

- Gromacs & LAMMPS
- DNN training acceleration with pytorch
 - [DeepBLAST](#)
- All pairs summation
 - [SkellySim](#) – CCB/SCC – Blackwell, Shelley, Kabacaoglu, Stein
 - [Pvfmm](#) – CCM – Malhotra

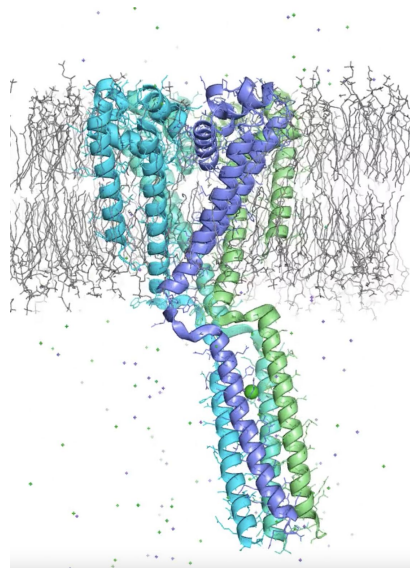
Software performance: GROMACS

- *Hand-optimized version by/for Nvidia GPUs: everything runs on the GPUs*
- *AMD GPU uses hipSYCL: some parts are still on the CPU!*

Ion Channel 150k atoms - Sonya Hanson CCB

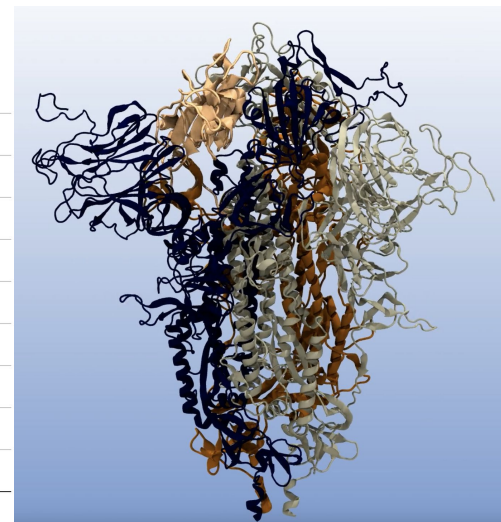
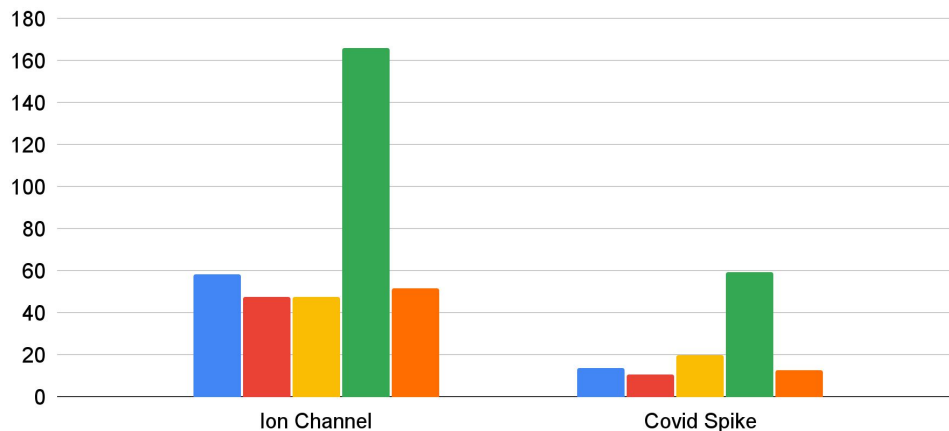
Covid Spike 750k atoms - Pilar Cossio CCM

Phu Tang CCM



GROMACS performance (ns of simulation / day: higher is better)

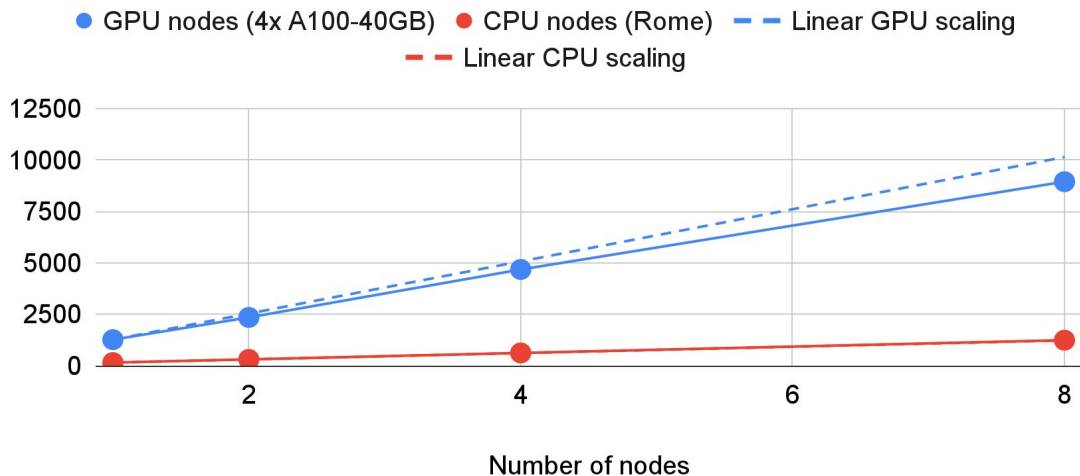
■ Rome single node ■ Icelake single node ■ A100 single GPU ■ A100 single node (4 GPUs)
■ AMD MI210 single GPU



Software performance: LAMMPS

- LAMMPS has several implementations that can use GPUs: the default one is mediocre. *We use the Kokkos one instead*
- Model: 3D Lennard-Jones liquid

LAMMPS Weak Scaling



Pytorch DNN Training

- *TM-Vec: template modeling vectors for fast homology detection and alignment* (Jamie Morton et al., in prep)
- “uses sequence alignments to learn structural features that can then be used to search for structure-structure similarities in large sequence databases”
- <https://github.com/flatironinstitute/deepblast>

Pytorch DNN Training (2)

Differentiable Needleman-Wunsch algorithm as loss function

Needleman-Wunsch

match = 1 mismatch = -1 gap = -1

		G	C	A	T	G	C	G
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

- Original algorithm discrete (uses 'max')
 - not differentiable – no traceback/backward pass
 - Modified to use “softmax” for training
- Algorithm is $O(m*n)$, i.e. the product of the two sequence lengths
- Single matrix calculation is serial, but you can do a lot of them at once!

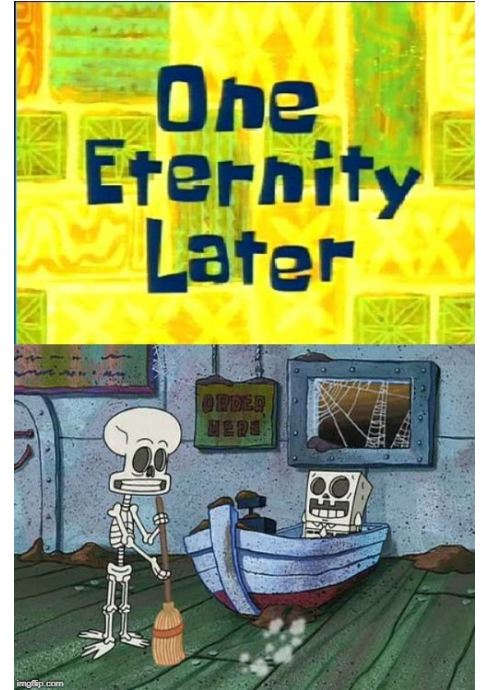
https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm

Pytorch DNN Training (3)

- All versions are calculated with python
 - no C/C++ code
- Pure python implementation
 - years to complete training
 - Hard to verify that it was even doing the right thing...

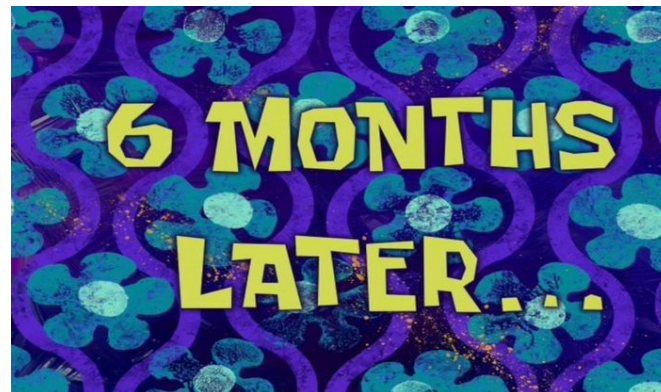
<https://github.com/flatironinstitute/deepblast/blob/master/deepblast/nw.py>

https://github.com/flatironinstitute/deepblast/blob/master/deepblast/nw_cuda.py



Pytorch DNN Training (3)

- Numba-CPU
 - About a day or two of coding/testing
 - Direct modification of original code
 - Months of training time
 - ~20x faster on 40-core skylakes



<https://github.com/flatironinstitute/deepblast/blob/master/deepblast/nw.py>





Pytorch DNN Training (4)

- Numba-CUDA
 - About a week of coding/optimizing
 - Days of training time
 - ~10x faster than numba-cpu version on v100
 - Slightly more fragile
 - Max sequence size limited
 - Harder to debug, all objects on GPU
 - “Easy” to support multi-node + multi-gpu



https://github.com/flatironinstitute/deepblast/blob/master/deepblast/nw_cuda.py

Pytorch DNN Training (5)

- Conclusion – weeks to save years
 -  Low precision OK
 -  Many small independent problems
 -  Enough parallel problems to saturate compute
 -  Low development effort

Without a question: worth the effort!

All pairs summation

$$\mathbf{U}_j = \sum_i^{N_{\text{src}}} \mathbf{G}(\mathbf{r}_{\text{trg},j} - \mathbf{r}_{\text{src},i})$$

- AKA the N-Body problem
- Embarrassingly parallel
 - Ideal for GPU calculations?
- Fast libraries for CPU approximations (Ewald, FMM, etc... see: CCM)
 - Often only ideal for large systems

All pairs summation (2)

Motivations:

1. FMM overkill/slow for small systems
2. Not all kernels needed are in FMM
3. Single point evaluations (streamlines)
4. No baseline to compare against
5. Generally useful for other projects
6. Curiosity!



SkellySim

FLATIRON
INSTITUTE
Scientific Computing Core

All pairs summation (3)

A naive implementation on CPU:

```
float all_pairs(const point_vec &r_src, const point_vec &r_trg) {  
    float ans = 0;  
    for (const auto &rt : r_trg)  
        for (const auto &rc : r_src)  
            ans += G(rt, rc);  
  
    return ans;  
}
```

All pairs summation (4)

A naive implementation on GPU**:

```
template <typename kernel>
__global__ void untiled_driver(const typename kernel::floattype *r_src, const typename kernel::floattype *r_trg,
                             typename kernel::floattype *__restrict__ u_trg, const typename kernel::floattype *f_src,
                             int n_src, int n_trg) {
    const int threadIdx = threadIdx.x;
    const int blkIdx = blockIdx.x;
    const int blkDim = blockDim.x;
    const int i_trg = threadIdx + blkIdx * blkDim;

    if (i_trg >= n_trg)
        return;

    for (int i = 0; i < kernel::trgdim; ++i)
        u_trg[i_trg * kernel::trgdim + i] = 0.0;

    for (int i_src = 0; i_src < n_src; ++i_src)
        kernel::uKernel(&r_src[i_src * 3], &r_trg[3 * i_trg], &f_src[i_src * kernel::srcdim],
                       &u_trg[i_trg * kernel::trgdim]);

    for (int i = 0; i < kernel::trgdim; ++i)
        u_trg[i_trg * kernel::trgdim + i] *= kernel::scale_factor;
}
```

- ** Not a totally fair comparison! This is obviously more generic...
- ** Does not include the memory copy code

All pairs summation (5)

```
template <typename kernel>
__global__ void tiled_driver(const typename kernel::floattype *r_src, const typename kernel::floattype *r_trg,
                           typename kernel::floattype *u_restrict, const typename kernel::floattype *f_src,
                           int n_src, int n_trg, int n_tiles) {

    using T = typename kernel::floattype;
    int i_trg = blockIdx.x * blockDim.x + threadIdx.x;

    extern __shared__ char shared_char[];
    T *shared = (T *)shared_char;
    const int shared_row_size = 3 + kernel::srcdim;
    const int toffset = shared_row_size * threadIdx.x;
    T *r_shared = (T *)(&shared[toffset]);
    T *f_shared = (T *)(&shared[toffset] + 3);
    if (i_trg < n_trg) {
        for (int i = 0; i < kernel::trgdim; ++i)
            u_trg[i_trg * kernel::trgdim + i] = 0.0;
    }
    for (int tile = 0; tile < n_tiles; tile++) {
        const int i_src = (tile * blockDim.x + threadIdx.x);
        for (int i = 0; i < 3; ++i)
            r_shared[i] = r_src[i_src * 3 + i];

        for (int i = 0; i < kernel::srcdim; ++i)
            f_shared[i] = f_src[i_src * kernel::srcdim + i];

        __syncthreads();

        // Loop over particles in our tile. But if tile contains i_src >= n_src, don't include those
        const int n_local_max = ((tile + 1) * (blockDim.x) > n_src) ? n_src - tile * blockDim.x : blockDim.x;
        if (i_trg < n_trg) {
            for (int i_local = 0; i_local < n_local_max; i_local++) {
                kernel::uKernel(&shared[shared_row_size * i_local], &r_trg[i_trg * 3],
                               &shared[shared_row_size * i_local + 3], &u_trg[i_trg * kernel::trgdim]);
            }
        }
        __syncthreads();
    }

    if (i_trg >= n_trg)
        return;

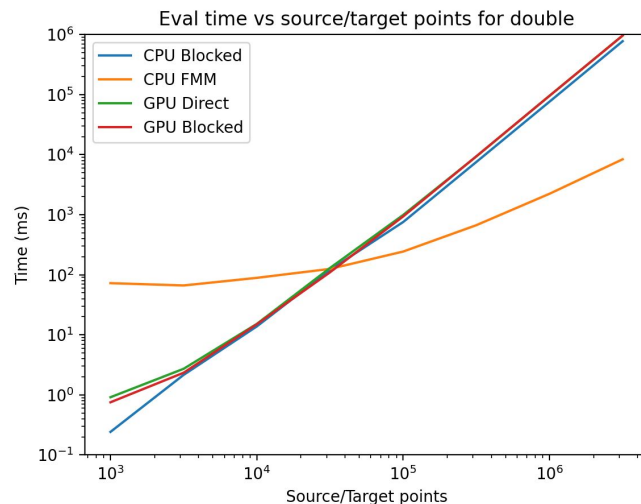
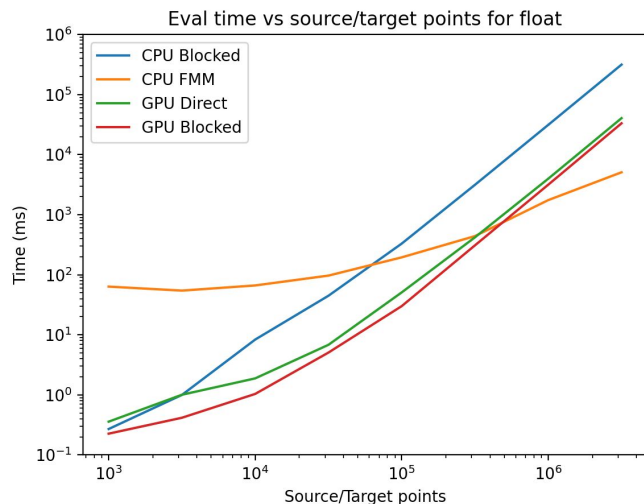
    for (int i = 0; i < kernel::trgdim; ++i)
        u_trg[i_trg * kernel::trgdim + i] *= kernel::scale_factor;
}
```

- Memory coalescence
- ~1.3x faster than naive
- Probably still far from optimal

All pairs summation (6)

Stokeslet:
$$\mathbf{G}(\mathbf{r}) = \frac{1}{8\pi\mu} \frac{\mathbf{I} + \hat{\mathbf{r}}\hat{\mathbf{r}}}{\|\mathbf{r}\|}$$

Quadro RTX 6000
2x Xeon Gold 6128 (skylake) @3.4GHz (6 cores x 2)

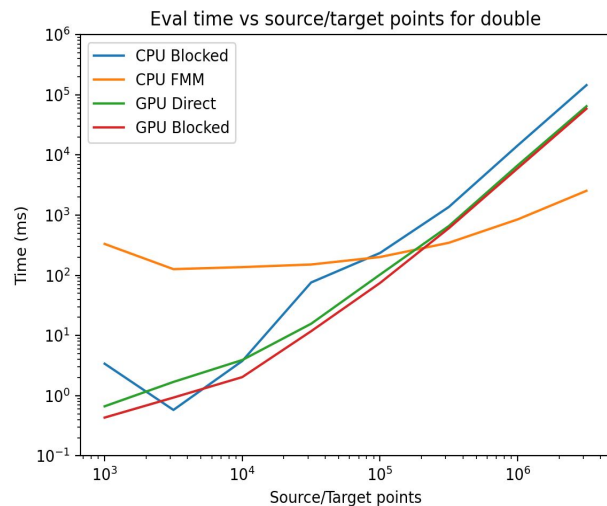
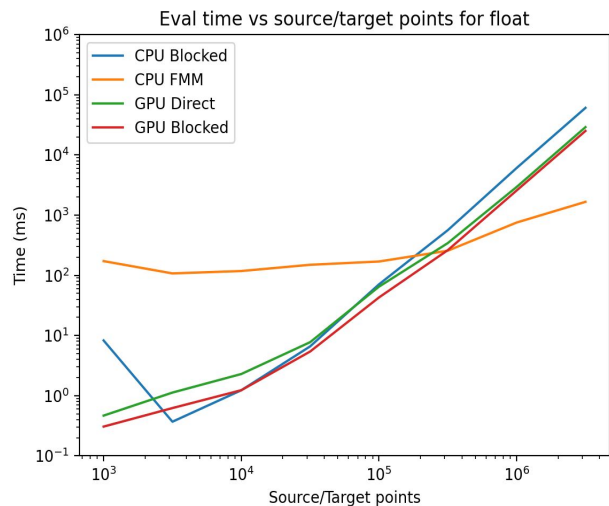


All pairs summation (7)

Stokeslet:
$$\mathbf{G}(\mathbf{r}) = \frac{1}{8\pi\mu} \frac{\mathbf{I} + \hat{\mathbf{r}}\hat{\mathbf{r}}}{\|\mathbf{r}\|}$$

NVIDIA A100-SXM

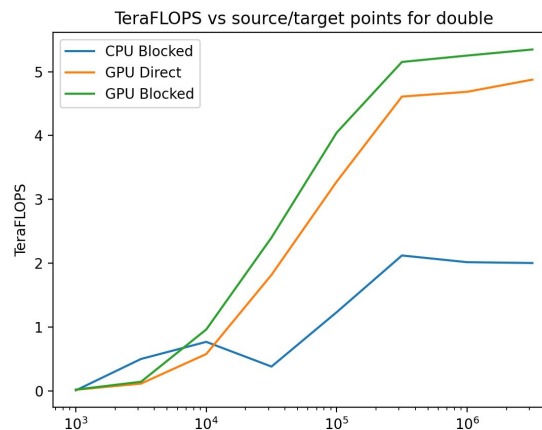
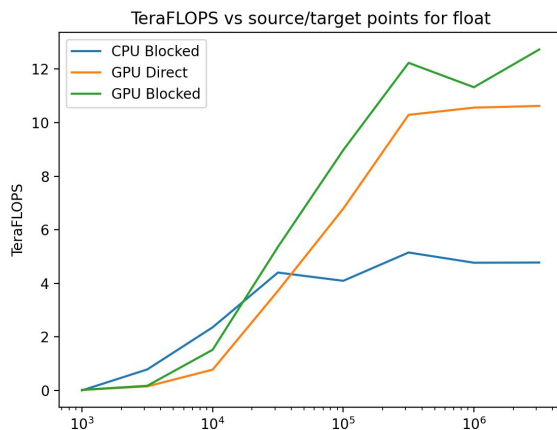
2x Xeon Platinum 8358 (icelake) @2.6 GHz (32 cores x 2)



All pairs summation (8)

Theoretical for floats:
CPU ~3.5 Tflops -> ~100%
GPU ~19.5 Tflops -> ~65%

Theoretical for doubles:
CPU ~1.75 Tflops -> ~100%
GPU ~9.7 Tflops -> ~55%









NVIDIA A100-SXM

2x Xeon Platinum 8358 (icelake) @3.4 GHz (32 cores x 2)

All pairs summation (9)

- Conclusion

-   Low precision (limited workstation power for doubles)
-  Many small independent problems
-   Enough parallel problems to saturate compute
-  Low development effort

Mixed – worth it in special cases

Needs more work for smaller systems

All pairs summation (10)

Further possible improvements:

- Cache memory (avoid extra mallocs on cuda device)
- Memory pinning
- Force better alignment
- Better blocking strategies? Block size?
- Tensor cores?
- Just talk to the Astro community :)
- Streaming
 - Overlaps memory transfers with calculation
- Hybridize algorithms**
 - FMM + GPU

** Actually in the works with Dhairya Malhotra's pvfmm

Getting the best possible performance

Nvidia Datacenter GPU	Nvidia A100 SXM	Nvidia H100 SXM	Nvidia H100 PCIe
GPU codename	GA100	GH100	GH100
GPU architecture	Ampere	Hopper	Hopper
GPU board form factor	SXM4	SXM5	PCIe Gen5
Launch date	May 2020	March 2022	March 2022
GPU process	TSMC 7nm N7	custom TSMC 4N	custom TSMC 4N
Die size	826mm ²	814 mm ²	814 mm ²
Transistor Count	54 billion	80 billion	80 billion
FP64 CUDA cores	3,456	8,448	7,296
FP32 CUDA cores	6,912	16,896	14,592
Tensor Cores	432	528	456
Streaming Multiprocessors	108	132	114
Peak FP64	9.7 teraflops	30 teraflops	24 teraflops
Peak FP64 Tensor Core	19.5 teraflops	60 teraflops	48 teraflops
Peak FP32	19.5 teraflops	60 teraflops	48 teraflops
Peak FP32 Tensor Core	156 teraflops 312 teraflops*	500 teraflops 1,000 teraflops*	400 teraflops 800 teraflops*
Peak BFLOAT16 Tensor Core	312 teraflops 624 teraflops*	1,000 teraflops 2,000 teraflops*	800 teraflops 1,600 teraflops*
Peak FP16 Tensor Core	312 teraflops 624 teraflops*	1,000 teraflops 2,000 teraflops*	800 teraflops 1,600 teraflops*
Peak FP8 Tensor Core	-	2,000 teraflops 4,000 teraflops*	1,600 teraflops 3,200 teraflops*
Peak INT8 Tensor Core	624 TOPS 1,248 TOPS*	2,000 TOPS 4,000 TOPS*	1,600 TOPS 3,200 TOPS*
Peak INT4 Tensor Core	1,248 TOPS 2,496 TOPS*	-	-
Interconnect	NVLink: 600GB/s PCI Gen4: 64GB/s	NVLink: 900GB/s PCI Gen5: 128GB/s	NVLink: 600GB/s PCI Gen5: 128GB/s
Max TDP	400 watts	700 watts	350 watts

*Effective TFLOPS or FLOPS using the Sparsity feature

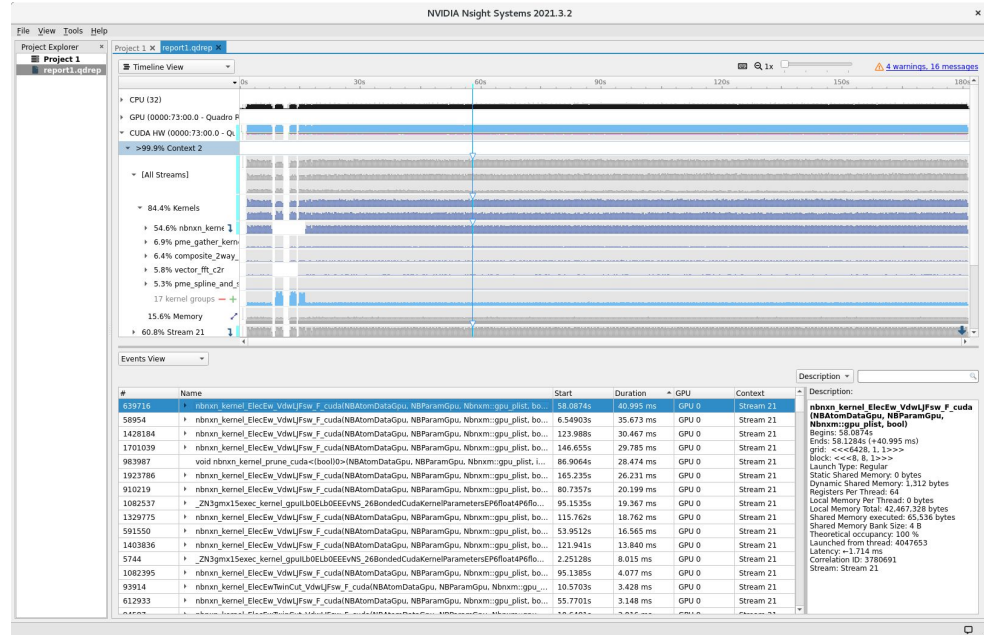
- If you can use cu* libraries, use them
- In CUDA, you need to use them explicitly (wrappers around ASM)
 - They can only do one thing: MMA
 - $D = A \times B + C$
 - Small matrices $\sim O(8 \times 8)$

- FP64: 2x in Tensor Core?
- FP32: > 8x in Tensor Core?

How to get the Tensor Core teraflops?

Tools to help programmers

- Nsight and Nvprof provide low level information to CUDA developers:
 - Kernel performance
 - Memory pattern issues
 - CPU ↔ GPU communications
- Profiler data can be imported by Tau (useful when mixing MPI, CUDA, OpenMP)



Portability

Between generations of GPUs

- Architectures evolve quickly
- Old optimizations get broken
- With GPUs:
 - No or bad optimizations often means slow code

Between models of the same generation

- Data center vs workstation cards
 - FP64 is capped on workstations
 - Memory speeds: HBM vs GDDR

Between vendors

- Different languages
 - CUDA (Nvidia)
 - hip/ROCm (AMD)
 - SYCL (standard, used by Intel)
- Different architectures: different optimizations
 - Tensor vs Matrix cores?
 - Memory hierarchy

Portability (2): generic frameworks

There are at least some attempts at standardization!

SYCL (syntactic sugar for OpenCL)

- Can be compiled as CUDA, HIP, OpenMP
- Pushed by Intel in OneAPI
- Data transfers are transparent
- Mix of regular C and C++ lambdas
- Low level optimizations are not portable

OpenACC

- Uses directives (`#pragma`) like OpenMP
- Built with Nvidia/PGI compiler
- Bugs in the compiler!

HIP (AMD high level language)

- Can be compiled as CUDA
- Source-to-source CUDA→HIP translator
- Low level optimizations are not portable!

Kokkos Framework

- Provides a parallel execution environment
- + architecture optimized libraries (linear algebra, graph)
- Can be overwhelming to start with

The Good, the Bad, and the Ugly?

The Good:

- Great performance
- Optimized vendor libraries
- More and more optimized codes

The Bad:

- Programming them directly is hard
- Getting their peak performance is harder (and requires ASM)

The Ugly:

- Getting portable performance is very hard
- For 10+ years, a single vendor has been preeminent, and they don't like Open Source

Should you use GPUs?

- The “absolutely!” cases:
 - You are using ML, or programs optimized for GPUs
 - You can use the BLAS, FFT, DNN libraries provided by the vendors
- The “maybe” cases:
 - Your code uses dense linear algebra
 - You are patient enough to reorganize it by blocks
- The “hum...” cases:
 - Your code is highly irregular (eg: graph)
 - Your project needs to be finished in 2 weeks

Resources at FI

Hardware

- 60 Nvidia V100 nodes*
- 72 Nvidia A100 nodes*
- 20 Nvidia H100 nodes*
- 2 AMD Instinct MI210 nodes**

*: 4 GPUs per node

** : 3 GPUs per node

Software in modules

- CUDA (and CUDA-aware OpenMPI)
- Nvidia/PGI compiler
- SYCL through Intel OneAPI
- HIP (module load rocm)
- hipSYCL
- LLVM

- Tools already configured for GPUs:
 - BLAS, FFT, NN, tensor libraries
 - PyTorch
 - GPU-aware OpenMPI