CENTRE OF EXCELLENCE IN SIMULATION OF WEATHER AND CLIMATE IN EUROPE

ESiWACE3 - Excellence in Simulation of Weather and Climate in Europe, Phase 3

GPU Optimization with Kernel Tuner

September 13, 2024

EuroHPC Joint Undertaking

Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) under grant agreement No 101093054.

Co-funded by the European Union

netherlands eScience center

Outline for the day

10:00	–	10:15	Welcome
10:15	–	10:45	Energy efficient GPU computing
10:45	–	11:30	Code optimizations for energy efficiency
11:30	–	11:45	Break
11:45	–	12:15	Mixed-precision programming techniques
12:15	–	12:45	Optimizing GPU core clock frequency
12:45	–	12:30	Q&A

Administrative announcements

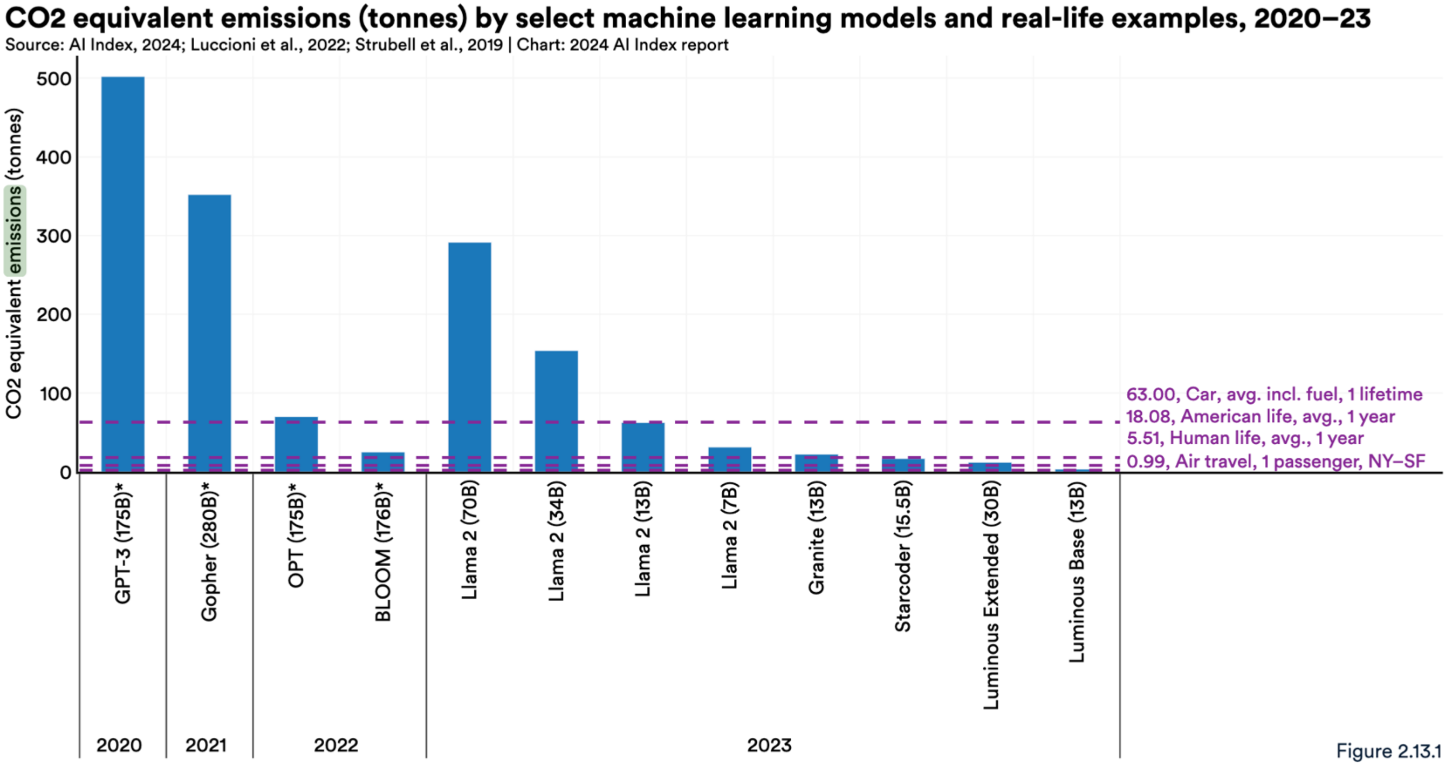
- We will have four sessions in which we start with introducing some new concepts and follow with a hands-on exercise
- The hands-on exercises include example kernels, but you are also welcome to experiment with your own code
- We will use JupyterHub hosted by the VSC for the hands-on, so you don't need to have a GPU or install anything locally
- You can download the slides here:
  - https://github.com/KernelTuner/kernel_tuner_tutorial/tree/master/slides/2024_VSC_ESiWACE3

Learning objectives

- Understand the energy footprint of computing
- Optimize applications for performance to reduce energy consumption
- Reduce data movement with mixed-precision techniques
- Tune GPU core frequencies to find the most energy-efficient configuration

Energy Efficient GPU Computing

LLM Training emissions



Source: Stanford AI Index report 2024

What is 500 tons of CO_2?

Roughly equal to:

- 8,268 tree seedlings grown for 10 years
- $80,000 in electricity bill
- 63 homes' energy use for a year in the US
- 111 passenger cars driving around for a year in the US

- Less than 2 days of running the Frontier supercomputer …

https://www.epa.gov/energy/greenhouse-gas-equivalencies-calculator esiwace

CENTRE OF EXCELLENCE IN SIMULATION OF WEATHER AND CLIMATE IN EUROPE

Energy cost of supercomputers

Frontier: #1 in TOP500 list (Jun 2024)

- #13 Green500 (Jun 2024)
- 20 Megawatt continuously
- $40 million annual electricity bill
- 100,000 metric tons of CO_2 annually
- ~20,000 cars on the road for a year in US

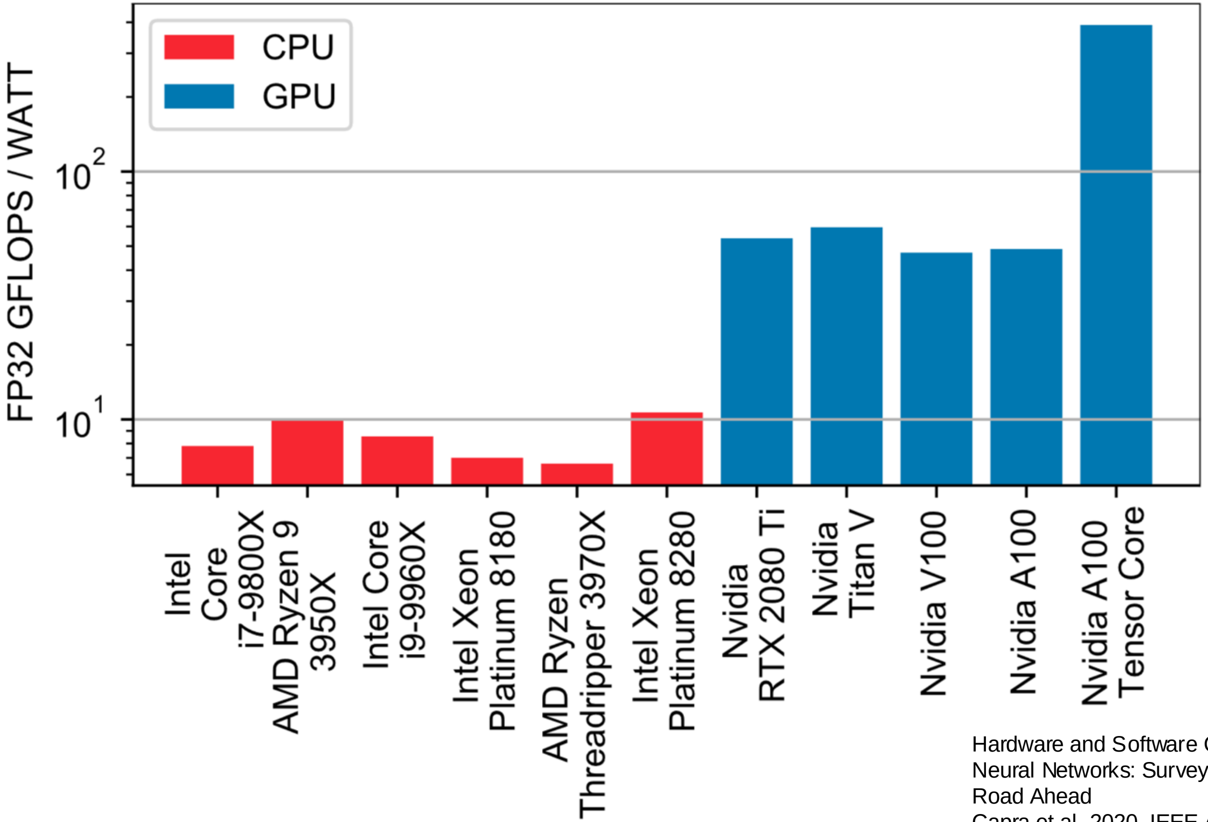
Summit: (#9, Frontier's predecessor)

- 64% of energy is consumed by GPUs

Efficient Computation through Tuned Approximation
David Keyes, SIAG/SC Supercomputing Spotlights 2022

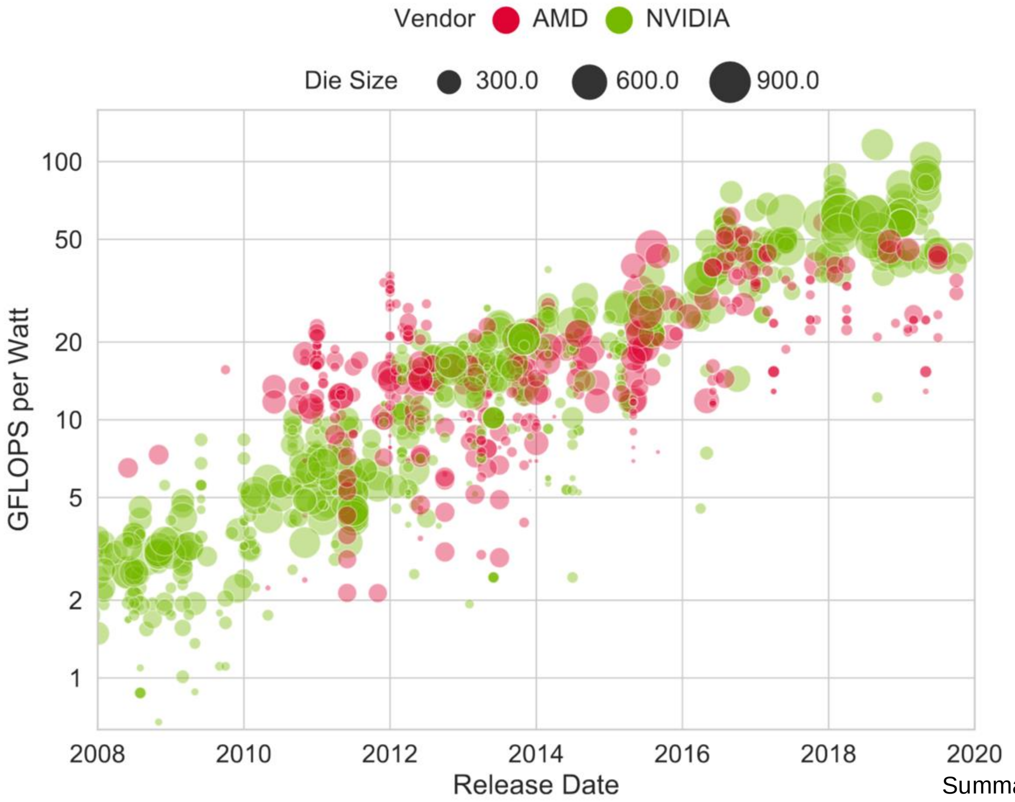
Autotuning based on frequency scaling toward energy efficiency of blockchain algorithms on graphics processing units M. Stachowski, A. Fiebig, and T. Rauber, Journal of Supercomputing, 2020.

GFLOPs/W for different architectures



Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead
Capra et al. 2020 IEEE Access

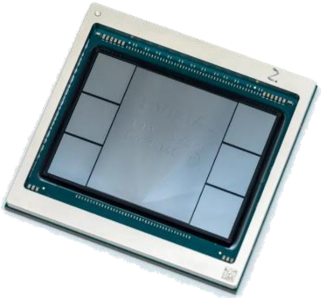
Energy Efficiency of GPUs



Summarizing CPU and GPU Design Trends
with Product Data
Sun et al. 2020

Energy, Heat, and Surface Size

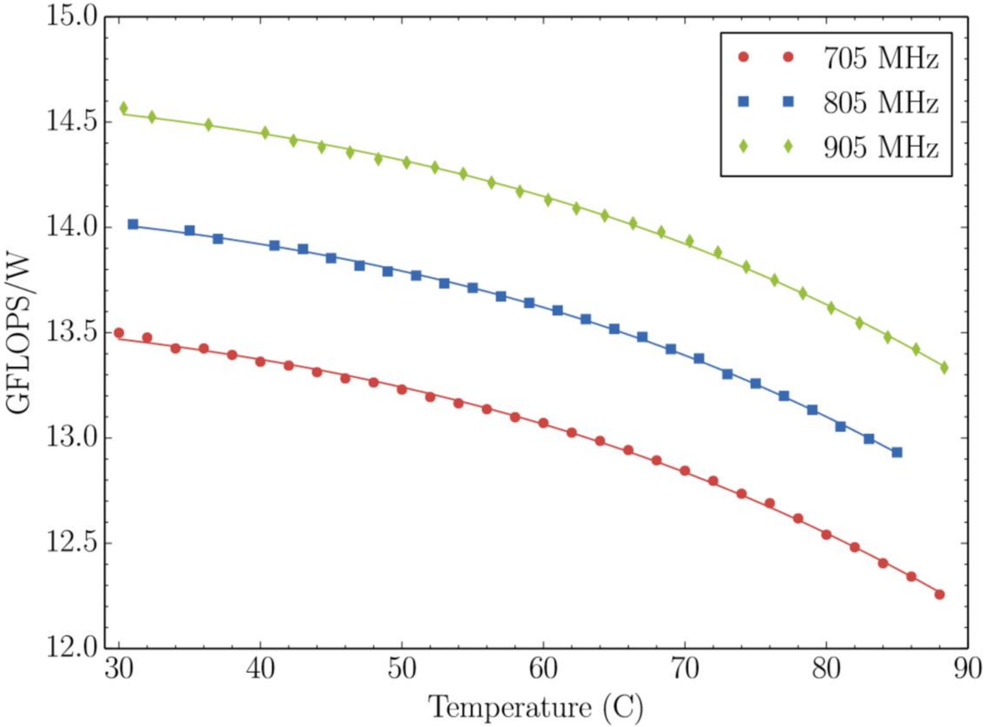
- Nvidia H100 GPU:
  - Energy: 350 Watt
  - Surface: 8.14 cm^2
  - Heat dissipation: 43.0 Watt/cm^2



- Light bulb:
  - Energy: 100 Watt
  - Surface: 15 cm^2
  - Heat dissipation: 6.7 Watt/cm^2

- Electric cooker:
  - Energy: 1800 Watt
  - Surface: 1017 cm^2
  - Heat dissipation: 1.8 Watt/cm^2

GPU Temperature – Energy Efficiency relation



Hotter GPUs can be ~7% less energy efficient

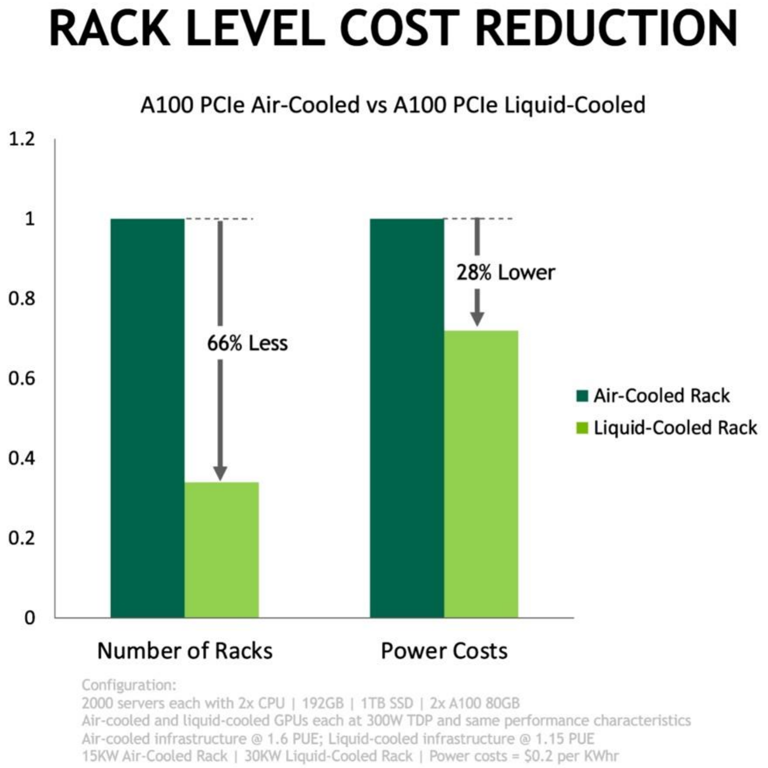
Results obtained with xGPU (radio astronomy correlator) on Nvidia K20

Optimizing performance-per-watt on GPUs in high performance computing
Price et al. 2016

What about cooling?

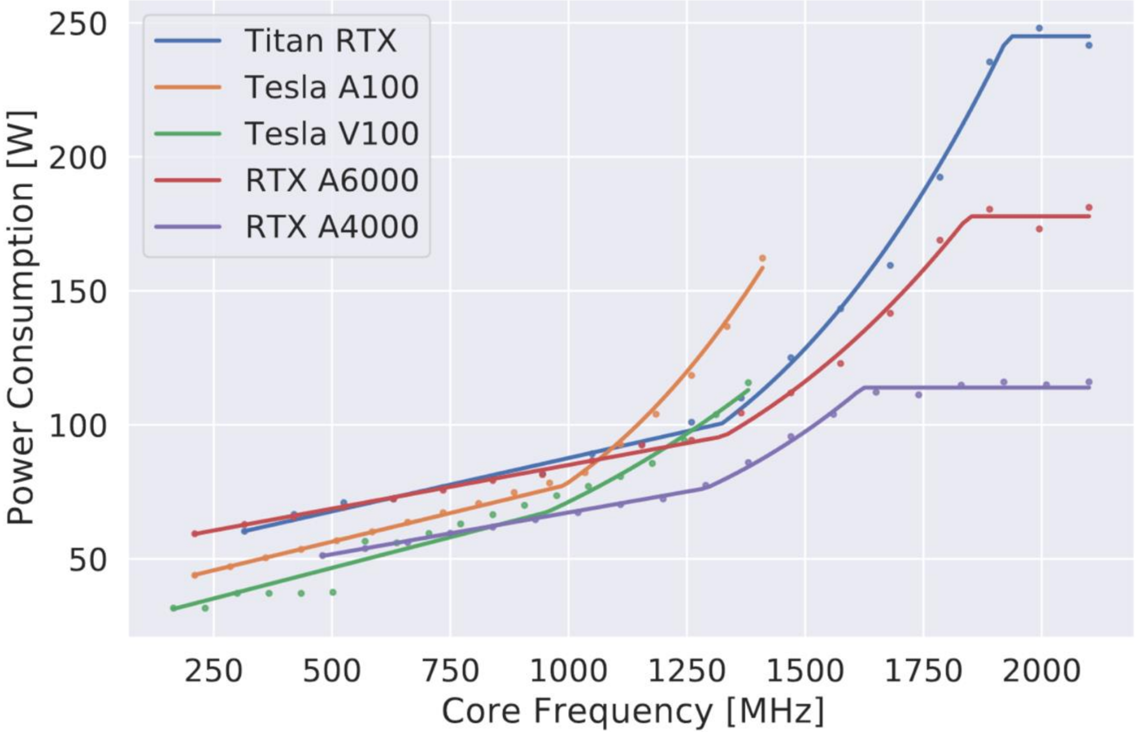
- Liquid cooling is more energy friendly than air cooling

- But as the efficiency difference between hot and cold GPUs is ~7%, you probably shouldn't overdo the cooling



https://blogs.nvidia.com/blog/2022/05/23/liquid-cooled-gpus-computex/

Clock frequency power relation



Going Green: optimizing GPUs for energy efficiency through model-steered auto-tuning
Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
PMBS workshop at SC22 2022

How is energy spent within a GPU?

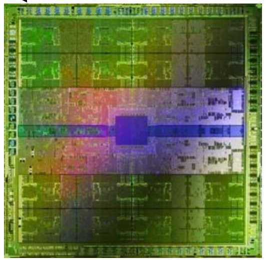
Moving data around is 20x more expensive than computing on it

Estimations for Nvidia H100:

- A single double-precision Fused Multiply-Add[1]: 13.7 pJ
- Moving the operands (4x 64-bits) for 10 mm within chip[2]: 294.4 pJ (21x more energy)

```
mad.f64  %f1, %f2, %f3, %f0; // c += a*b;
```

1 cm



[1]Based on 25.6 TFLOP/s peak at 350 Watt TDP
https://www.nvidia.com/en-us/data-center/h100/

[2]Based on 115 femtojoule/bit/mm
GPUs and the Future of Parallel Computing
Keckler et al. 2011

How do we create energy efficient GPU applications?

Three strategies for energy efficient GPU Computing:

1. Use for shorter amount of time
2. Minimize data movements
3. Optimize device settings

How do we create energy efficient GPU applications?

Three strategies for energy efficient GPU Computing:

1. Use for shorter amount of time → Optimize application performance
2. Minimize data movements → Lower/mixed precision techniques
3. Optimize device settings → Optimize clock frequency

Code Optimizations for Energy Efficiency

GPU code optimizations

- Modify the kernel source code to improve performance or tunability
- Effects on performance can be different on different GPUs or different input data
- You can tune:
  - Enabling or disabling an optimization
  - The parameters introduced by certain optimizations
- You often need to combine multiple different optimizations with specific tunable parameter values to arrive at optimal performance

Further reading

- In March 2023, we published a literature review summarizing the last decade of code optimizations for GPU programming
  - We describe which optimizations are used in literature and how they are used

- *Optimization Techniques for GPU Programming*
  Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri Bal
  ACM Computing surveys 2023
  https://dl.acm.org/doi/abs/10.1145/3570638

Overview of GPU Optimizations

- Coalescing memory accesses
- Host/device communication
- Kernel fusion
- Loop blocking
- Loop unrolling
- Prefetching
- Recomputing values
- Reducing atomics
- Reducing branch divergence
- Reducing redundant work
- Reducing register usage
- Reformatting input data
- Using a specific memory space
- Using warp shuffle instructions
- Varying work per thread
- Vectorization

Overview of GPU Optimizations

- Coalescing memory accesses
- Host/device communication
- **Kernel fusion**
- **Loop blocking**
- Loop unrolling
- Prefetching
- Recomputing values
- Reducing atomics
- Reducing branch divergence
- Reducing redundant work
- Reducing register usage
- Reformatting input data
- Using a specific memory space
- Using warp shuffle instructions
- Varying work per thread
- Vectorization

Kernel fusion

Merge one or more kernels into one kernel

- Why?
  - Reduces data movements between off-chip DRAM and GPU registers
  - Moving data around is more expensive than computing on it
- How?
  - Fuse the kernel arguments and computations of two kernels into one
  - Demote a kernel to a `__device__` function and call it from another kernel
  - *Temporal fusion:* merge multiple calls of the same kernel into one

Kernel fusion

```
// c = a+b
vector_add<<<grid, threads>>>(c, a, b, n);
// e = c+d
vector_add<<<grid, threads>>>(e, c, d, n);

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

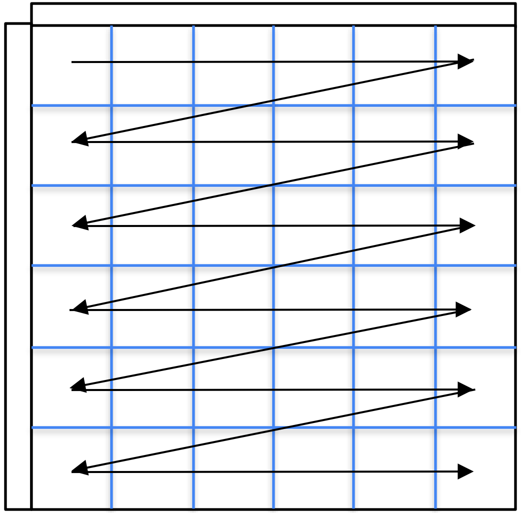
```
// e = a+b+d
vector_3add<<<grid, threads>>>(e, a, b, d, n);

__global__
void vector_3add(float *d, float *a, float *b,
                 float *c, int n) {
    int i = (blockIdx.x*blockDim.x)+threadIdx.x;
    if (i < n) {
        d[i] = a[i] + b[i] + c[i];
    }
}
```

Loop blocking

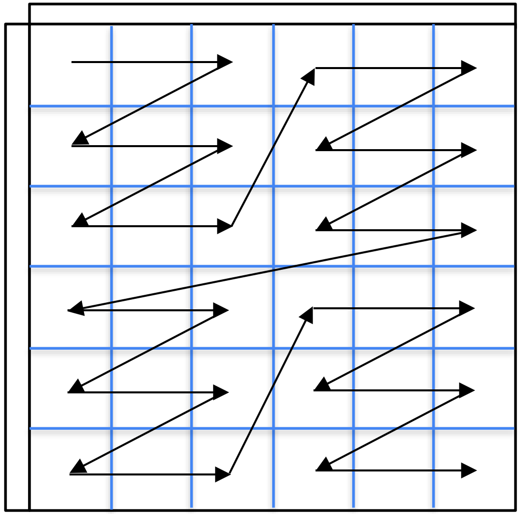
Modify the structure of one or more loops to work in blocks over the data

- Why?
  - Increases spatial / temporal locality
  - Reduces the ‘working set’ of the algorithm
- How?
  - Change the order of computations and data accesses in nested loops
  - Usually nearly doubles the number of for-loops in the code
  - Outer-loops iterate over the blocks
  - Inner-loops iterate within each block

Loop blocking



```
for (int j=0; j<ny; j++) {
    for (int i=0; i<nx; i++) {
        ...[j*nx + i]
    }
}
```



```
for (int j=0; j<ny; j+=nyb) {
    for (int i=0; i<nx; i+=nxb) {

        for (int jb=0; jb<nyb; jb++) {
            for (int ib=0; ib<nxb; ib++) {
                ...[(j+jb)*nx + (i+ib)]
            }
        }
    }
}
```

Hands-on

First hands-on

- The first hands-on notebook is:
  - https://github.com/KernelTuner/kernel_tuner_tutorial/blob/master/hands-on/esiwace3/04_Code_Optimizations_for_Energy.ipynb

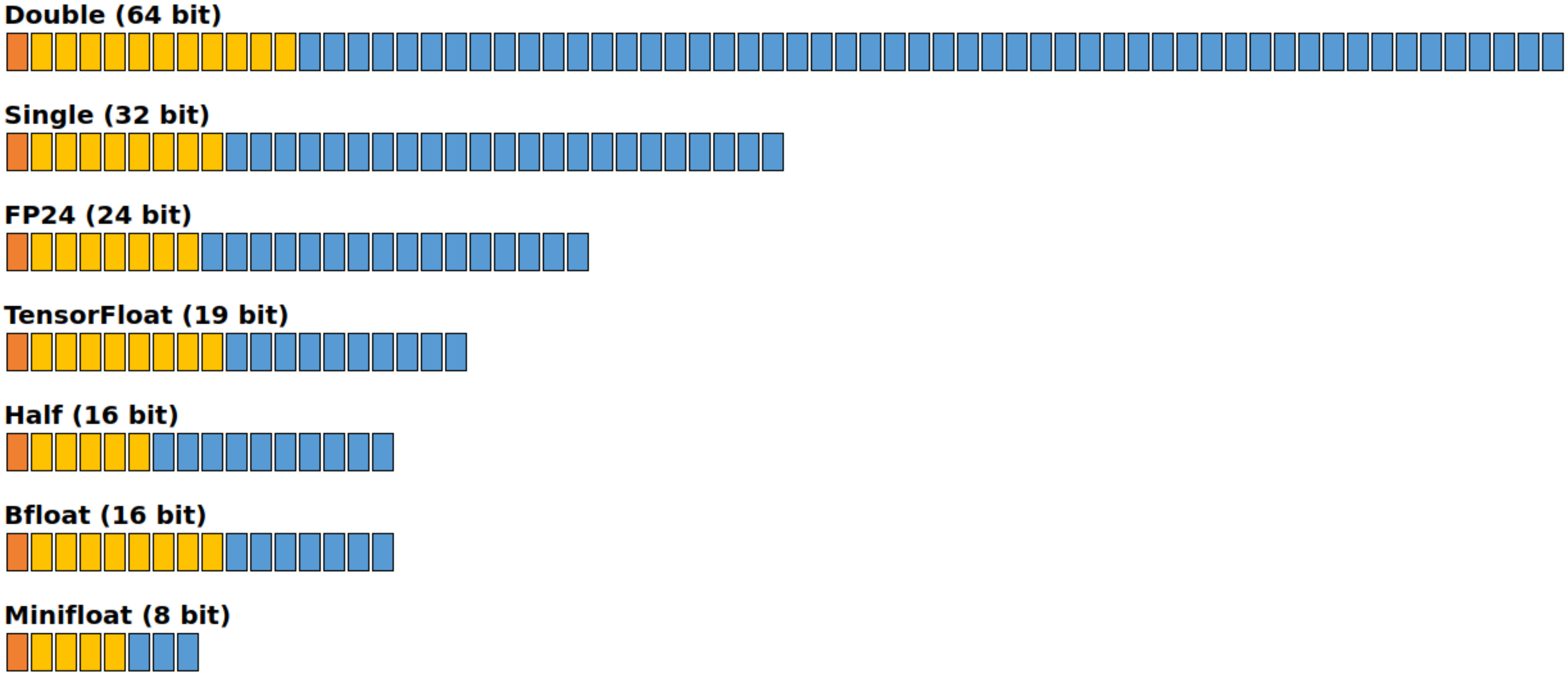
- The goal of this hands-on is to:
  - See an example of Kernel Fusion
  - Compare the energy consumption of different kernels

- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors

Mixed-Precision Programming Techniques

esiwace
CENTRE OF EXCELLENCE IN SIMULATION OF WEATHER AND CLIMATE IN EUROPE

Low precision computing



Low precision computing

- Low precision has many benefits ☺ !
  - Faster **computation**
    - Less compute cycles required, especially double precision is often slow
  - Lower **memory footprint**
    - Less bits required per number
  - Better **cache utilization**
    - Higher cache hit rates
  - Higher *effective* **memory bandwidth**
    - More numbers per second
  - Lower **register usage**
    - Increases GPU occupancy, thus performance
  - All these points also increase **energy efficiency**

Benefits of low precision computing

- But, at the cost of loss in precision ☹

Floating-point	Value of Pi	Error
Infinite bits	3.14159265358979323846264338327950288419716939937 5...	0
64 bit	3.141592653589793115997963468544185161590576171875	3.9×10^{-15} %
32 bit	3.14159250259399414062500000000000000000000000000 0	0.000005%
16 bit	3.140625000000000000000000000000000000000000000000	0.03%
8 bit	3.000000000000000000000000000000000000000000000000	4.5%

Mixed-precision arithmetic

- Core idea of **mixed precision:**
  - What if we **mix different** precision levels in one application?
  - Use **different floating-point types** for **different variables** in code
- Trade-off between **performance** and **numerical accuracy**
  - Ideally, we want to *maximize* performance while *minimizing* the error
- Creates **huge** search space
  - What precision should be used for each variable?
  - Example: 20 variables and 4 precisions gives 4^{20} = 1 trillions combinations!

Mixed-precision arithmetic

- **Core idea:**
  - What if we **mix different** precision levels in one application?
  - Use different floating-point types for different variables in code
- Leads to **trade-off** between **accuracy** and **performance**
  - **Lower precision** typically results in **higher performance**
  - Need to find **balance** between **error** and **speedup**
- What precision should be used for each variable?
  - Ideally, we want **maximum performance** for an **acceptable error**
  - Auto-tuning to the rescue!

Mixed-precision arithmetic

Core idea:

- What if we **mix different** precision levels in one application?

Examples:

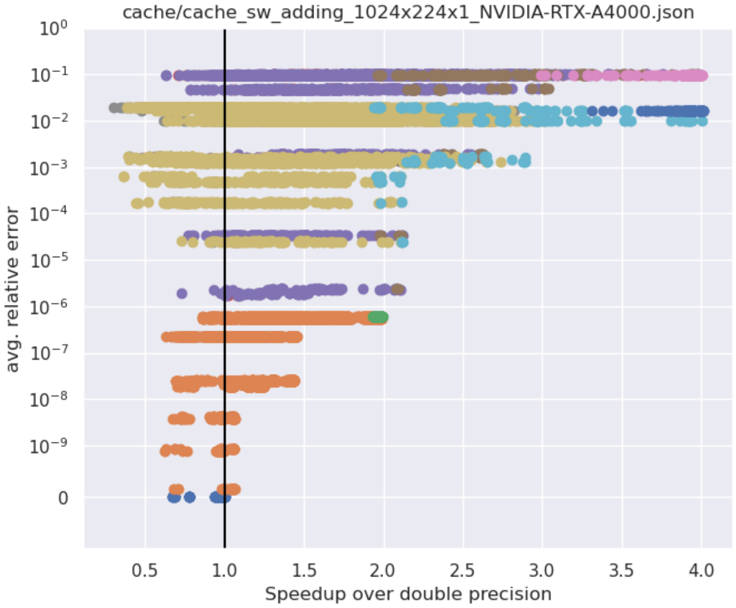
- Deep Learning
  - Commonly uses 16 or 8-bit (even 1 bit!) floating-point numbers
- Fluid dynamics simulations
  - High precision only for critical parts, such as turbulence modeling
- Molecular Dynamics
  - Lower precision for long-range calculations
- Finite Element Analysis
  - Iterative methods to solve large linear systems

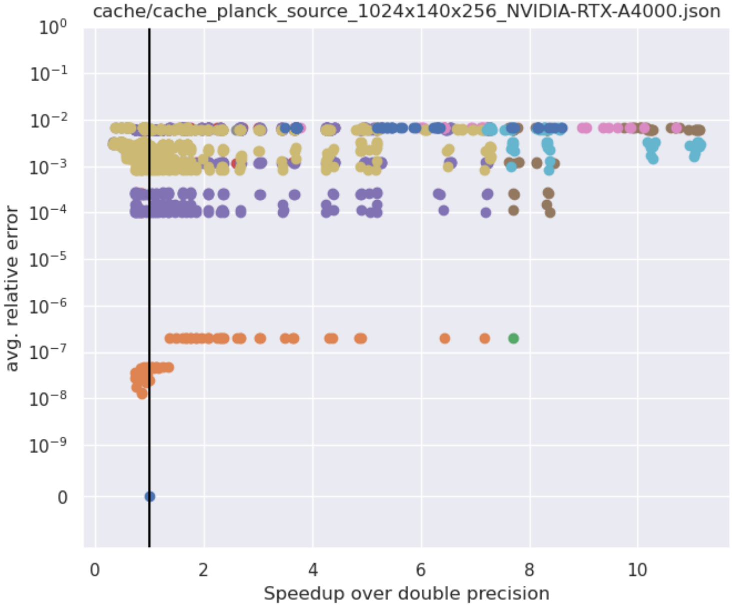
Example mixed-precision applications

Mixed precision balances **performance** and **numerical accuracy**

- Deep Learning
  - Commonly uses 16 or 8-bit (even 1 bit!) floating-point numbers
- Fluid dynamics simulations
  - High precision only for critical parts, such as turbulence modeling
- Molecular Dynamics
  - Lower precision for long-range calculations
- Finite Element Analysis
  - Iterative methods to solve large linear systems

Example radiation solver





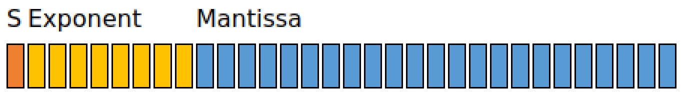
Floating-point format (IEEE 754)

- IEEE 754 standard is implemented in all architectures
- Floating-point number consists of three parts:
  - S: sign (+ or -)
  - M: mantissa/significand
  - E: exponent
- Floating-point number represented using exponential format:
  - $(-1)^S \times M \times 2^E$
  - Example: $+1.42 \times 2^3$ means S=+1, M=1.42, E=3
  - Where $1 \leq M < 2$, which makes representation unique
  - There are also non-normal numbers: NaN, Inf, subnormal

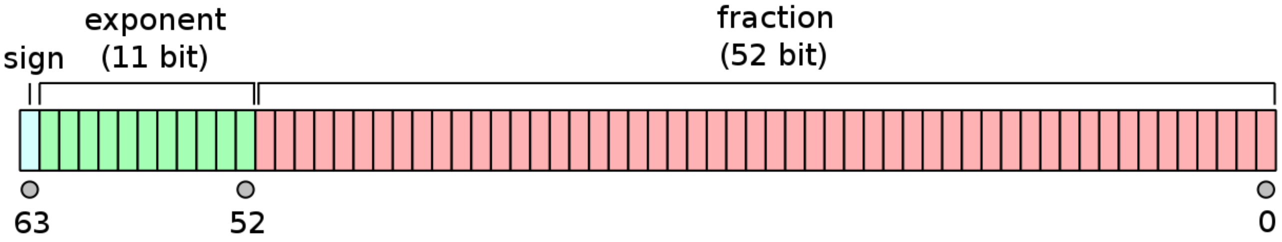
Floating-point format (IEEE 754)

- Sign bit (1 bit)
- Mantissa/significand (A bits)
  - Determines number of **significant digits**
  - Results rounded to number of decimal places
  - Example: A=23 means ~7 decimal places
- Exponent (B bits)
  - Determines **range** of numbers
  - Numbers outside range become *zero* or *infinity*
  - Example: B=8 means range is ~10^{-38} to ~10^{38}
- Total size: **1 + A + B bits**

Type	$\sqrt{2}$
A=52 (Float64)	1.41421356237309
A=23 (Float32)	1.414213
A=10 (Float16)	1.414
A=2 (Float8)	1.5



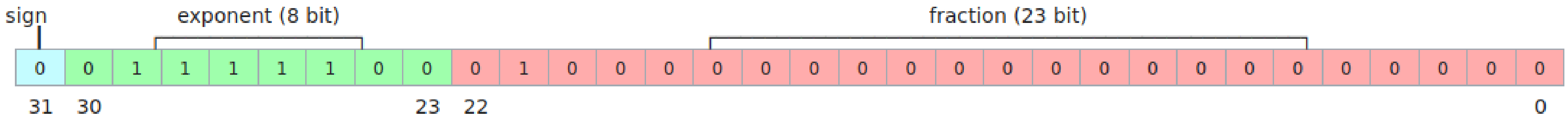
FP types: `double`



- **`double`** precision (64 bits) prevalent in scientific computing
- GPUs typically slow on double arithmetic
  - Except the scientific/datacenter-rated GPUs

Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
double	64	11	52	2.2e-308	1.8e+308	15	1 + 2.22e-16

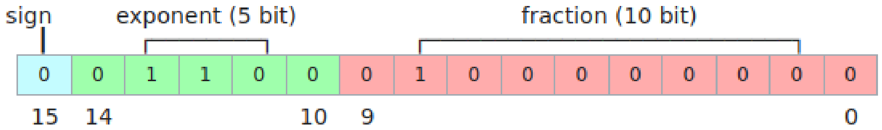
FP types: `float`



- **Single** precision (32 bits) balances accuracy and throughput
- Widely used in graphics and general GPU applications

Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
float	32	8	23	1.2e-38	3.4e+38	6	1.000000119

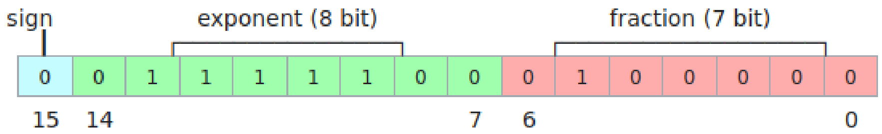
FP types: `half`



- Introduced with NVIDIA's Pascal architecture (2016)
- Double computational throughput of float
- Limited range, reasonable accuracy

Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
half	16	5	10	0.000061	65536	3	1.00097

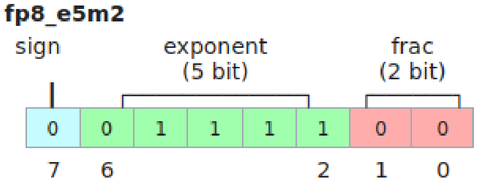
FP types: `bfloat`

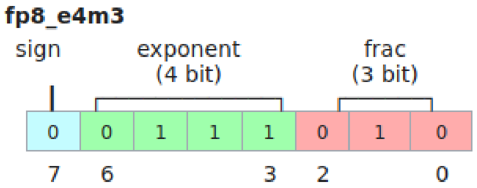


- "Brain" Floating-point. Introduced by Google Brain project
- Introduced with NVIDIA's Ampere architecture (2020)
- Large range, limited accuracy

Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
bfloat	16	8	7	1.2e-38	3.4e+38	2	1.00781

FP types: 8-bit floats





- Introduced with NVIDIA's Hopper architecture (2022)
- Two flavors: 5+2 bits or 4+3 bits
- No arithmetic functions, only conversions

Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
fp8_e4m3	8	4	3	0.015625	256	1	1.125
fp8_e5m2	8	5	2	0.000061	65536	0	1.25

Floating-point overview

Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
double	64	11	52	2.2e-308	1.8e+308	15	1 + 2.22e-16
float	32	8	23	1.2e-38	3.4e+38	6	1.000000119
half	16	5	10	0.000061	65536	3	1.00097
bfloat	16	8	7	1.2e-38	3.4e+38	2	1.00781
fp8_e4m3	8	4	3	0.015625	256	1	1.125
fp8_e5m2	8	5	2	0.000061	65536	<1	1.25

Mixed precision in practice for CUDA

- Create type aliases in kernels
  - C: use preprocess `#define`
  - C++: use template parameters
- Available data types in CUDA
  - `double` and `float` are predefined
  - `__half` found in `<cuda_fp16.h>`
  - `__nv_bfloat16` found in `<cuda_bf16.h>`
  - `__nv_fp8_eXmY` found in `<cuda_fp8.h>`

Example kernel

```
__global__ void vector_add(
        int n,
        const float* A,
        const float* B,
              float* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n)
      C[i] = A[i] + B[i];
}
```

Example kernel

```
#define TYPE_A float
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
        int n,
        const TYPE_A* A,
        const TYPE_B* B,
              TYPE_C* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n)
      C[i] = A[i] + B[i];
}
```

Example kernel

```
#include <cuda_fp16.h>


#define TYPE_A __half
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
        int n,
        const TYPE_A* A,
        const TYPE_B* B,
              TYPE_C* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n)
      C[i] = A[i] + B[i];
}
```

Example kernel

```
#include <cuda_fp16.h>


#define TYPE_A __half
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
        int n,
        const TYPE_A* A,
        const TYPE_B* B,
              TYPE_C* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n)
      C[i] = A[i] + B[i];
}
```

Does not compile! ☹

```
kernel.cu(15): error: no operator "+" matches
these operands
operand types are: __half + float
```

Example kernel

```
#include <cuda_fp16.h>


#define TYPE_A __half
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
        int n,
        const TYPE_A* A,
        const TYPE_B* B,
              TYPE_C* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n)
      C[i] = A[i] + B[i];
}
```

Does not compile! ☹

```
kernel.cu(15): error: no operator "+" matches
these operands
operand types are: __half + float
```

Mixed precision programing challenges

- No type promotion
  - Cannot mix types in binary operations
- Some operations require intrinsics
  - `__hdiv()`, `__hsin()`, `__hfmad()`
- Missing operations
  - No `__htan()`?
- Missing or awkward type conversion
  - `__nv_cvt_bfloat16raw2_to_fp8x2`
  - No `fp8` to `double`?
  - No `half` to `bfloat16`?

```
__global__ void kernel(const __half* input, float constant, float* output) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    __half in0 = input[2 * i + 0];
    __half in1 = input[2 * 1 + 1];
    __half2 a = __halves2half2(in0, int1);
    float b = float(constant);
    __half c = __float2half(b);
    __half2 d = __half2half2(c);
    __half2 e = __hadd2(a, d);
    __half f = __low2half(e);
    __half g = __high2half(e);
    float out0 = __half2float(f);
    float out1 = __half2float(g);
    output[2 * i + 0] = out0;
    output[2 * i + 1] = out1;
}
```

```
__global__ void kernel(const __half* input, float constant, float* output) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    __half in0 = input[2 * i + 0];
    __half in1 = input[2 * 1 + 1];
    __half2 a = __halves2half2(in0, int1);
    float b = float(constant);
    __half c = __float2half(b);
    __half2 d = __half2half2(c);
    __half2 e = __hadd2(a, d);
    __half f = __low2half(e);
    __half g = __high2half(e);
    float out0 = __half2float(f);
    float out1 = __half2float(g);
    output[2 * i + 0] = out0;
    output[2 * i + 1] = out1;
}
```

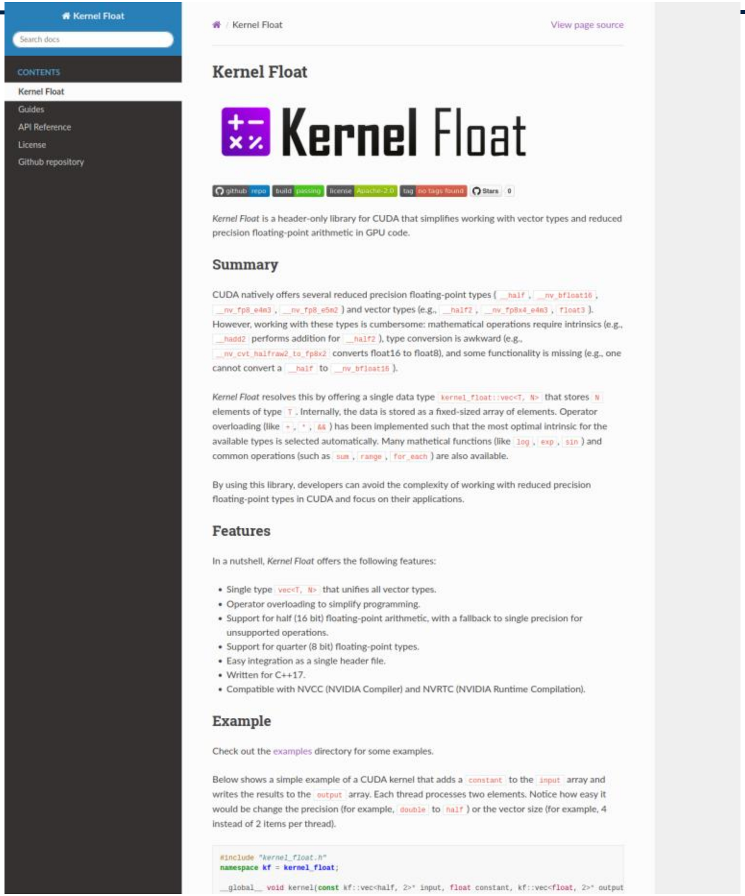
```
#include "kernel_float.h"
namespace kf = kernel_float;

__global__ void kernel(const kf::vec<half, 2>* input, float constant, kf::vec<float, 2>* output) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    output[i] = input[i] + kf::cast<half>(constant);
}
```

Kernel Float

https://github.com/KernelTuner/kernel_float

Kernel Float

- Header-only C++ library to simplify mixed precision GPU programming
- Offers single type: vec<T, N>
  - N elements of type T
  - Auto selects optimal storage format
- Offers all mathematical operations
  - Auto selects best intrinsic
  - Fallback to single precision for missing operations



Kernel Float

Search docs

CONTENTS

Kernel Float

Guides

API Reference

License

Github repository

Kernel Float / View page source

Kernel Float

Kernel Float

github repo | build passing | license Apache-2.0 | tag no tags found | Stars 0

Kernel Float is a header-only library for CUDA that simplifies working with vector types and reduced precision floating-point arithmetic in GPU code.

Summary

CUDA natively offers several reduced precision floating-point types (__half, __nv_bfloat16, __nv_fp8_e4m3, __nv_fp8_e5m2) and vector types (e.g., __half2, __nv_fp8x4_e4m3, float3). However, working with these types is cumbersome: mathematical operations require intrinsics (e.g., __hadd2 performs addition for __half2), type conversion is awkward (e.g., __nv_cvt_halfraw2_to_fp8x2 converts float16 to float8), and some functionality is missing (e.g., one cannot convert a __half to __nv_bfloat16).

Kernel Float resolves this by offering a single data type kernel_float::vec<T, N> that stores N elements of type T. Internally, the data is stored as a fixed-sized array of elements. Operator overloading (like +, *, &&) has been implemented such that the most optimal intrinsic for the available types is selected automatically. Many mathetical functions (like log, exp, sin) and common operations (such as sum, range, for_each) are also available.

By using this library, developers can avoid the complexity of working with reduced precision floating-point types in CUDA and focus on their applications.

Features

In a nutshell, *Kernel Float* offers the following features:

- Single type vec<T, N> that unifies all vector types.
- Operator overloading to simplify programming.
- Support for half (16 bit) floating-point arithmetic, with a fallback to single precision for unsupported operations.
- Support for quarter (8 bit) floating-point types.
- Easy integration as a single header file.
- Written for C++17.
- Compatible with NVCC (NVIDIA Compiler) and NVRTC (NVIDIA Runtime Compilation).

Example

Check out the examples directory for some examples.

Below shows a simple example of a CUDA kernel that adds a constant to the input array and writes the results to the output array. Each thread processes two elements. Notice how easy it would be change the precision (for example, double to half) or the vector size (for example, 4 instead of 2 items per thread).

```
#include "kernel_float.h"
namespace kf = kernel_float;

__global__ void kernel(const kf::vec<half, 2>* input, float constant, kf::vec<float, 2>* output
```

kerneltuner.github.io/kernel_float/

Example kernel

```
#define TYPE_A float
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
        int n,
        const TYPE_A* A,
        const TYPE_B* B,
              TYPE_C* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n)
      C[i] = A[i] + B[i];
}
```

Example kernel

```
#include "kernel_float.h"


#define TYPE_A float
#define TYPE_B float
#define TYPE_C float


__global__ void vector_add(
        int n,
        const kernel_float::vec<TYPE_A, 1>* A,
        const kernel_float::vec<TYPE_B, 1>* B,
              kernel_float::vec<TYPE_C, 1>* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n)
      C[i] = A[i] + B[i];
}
```

Example kernel

```
#include "kernel_float.h"


#define TYPE_A __half
#define TYPE_B float
#define TYPE_C float


__global__ void vector_add(
        int n,
        const kernel_float::vec<TYPE_A, 1>* A,
        const kernel_float::vec<TYPE_B, 1>* B,
              kernel_float::vec<TYPE_C, 1>* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n)
      C[i] = A[i] + B[i];
}
```

Vectorization

- Kernel Float automatically uses **vector** intrinsics
  - Requires using `kernel_float::vec<T, N>` with N ≥ 2
- Several types benefit from vectorization!
  - `half` and `bfloat` require vectorized intrinsics for high throughput
  - Vectorized memory operations
  - Vectorized integer operations
  - ...

Example kernel

```
#include "kernel_float.h"


#define TYPE_A float
#define TYPE_B float
#define TYPE_C __half
#define VECTOR_SIZE 1

__global__ void vector_add(
        int n,
        const kernel_float::vec<TYPE_A, VECTOR_SIZE>* A,
        const kernel_float::vec<TYPE_B, VECTOR_SIZE>* B,
              kernel_float::vec<TYPE_C, VECTOR_SIZE>* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i * VECTOR_SIZE < n)
      C[i] = A[i] + B[i];
}
```

Example kernel

```
#include "kernel_float.h"


#define TYPE_A float
#define TYPE_B float
#define TYPE_C __half
#define VECTOR_SIZE 2

__global__ void vector_add(
        int n,
        const kernel_float::vec<TYPE_A, VECTOR_SIZE>* A,
        const kernel_float::vec<TYPE_B, VECTOR_SIZE>* B,
              kernel_float::vec<TYPE_C, VECTOR_SIZE>* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i * VECTOR_SIZE < n)
      C[i] = A[i] + B[i];
}
```

Tuning Problem

- **Accuracy** vs **performance** trade-off
  - What type should we use for each variable?
  - Ideally want **high performance** with **low error**
- Variables datatypes and **kernel parameters** both affect performance
  - Usually heavily **intertwined**, we cannot tune them separately
- Leads to large **search-space**, for example:
  - 10 variables and 4 precision levels: 4^{10} = 1 million options
  - 8 parameters with each 6 options: 6^8 = 1 million options
  - Total: 1 trillion configurations!

Kernel Tuner

Kernel Tuner offers native support for *accuracy tuning*

- **Step 1**: Add tunable floating-point types as **tuning parameters**
- **Step 2**: Wrap inputs/outputs in `TunablePrecision` objects
- **Step 3**: provide **reference output** as `answer`
- **Step 4**: Add `AccuracyObserver`

See the example:

- examples/cuda/accuracy.py

Kernel Tuner Example

```
size = 100000000
n = numpy.int32(size)
a = numpy.random.randn(size)
b = numpy.random.randn(size)
c = numpy.zeros_like(b)

 args = [n,
     TunablePrecision("float_type", a),
     TunablePrecision("float_type", b),
     TunablePrecision("float_type", c)]

 answer = [None, None, None, a + b]

 tune_params = dict()
 tune_params["block_size_x"] = [32, 64, 128, 256, 512, 1024]
 tune_params["float_type"] = ["float", "double", "half"]

 observers = [AccuracyObserver("RMSE")]

 results, env = tune_kernel("vector_add",kernel_string,
     size, args, tune_params, answer=answer,
     observers=observers, lang="CUDA")
```

Kernel Tuner Example

```
size = 100000000
n = numpy.int32(size)
a = numpy.random.randn(size)
b = numpy.random.randn(size)
c = numpy.zeros_like(b)

 args = [n,
     TunablePrecision("float_type", a),
     TunablePrecision("float_type", b),
     TunablePrecision("float_type", c)]

 answer = [None, None, None, a + b]

 tune_params = dict()
 tune_params["block_size_x"] = [32, 64, 128, 256, 512, 1024]
 tune_params["float_type"] = ["float", "double", "half"]

 observers = [AccuracyObserver("RMSE")]

 results, env = tune_kernel("vector_add",kernel_string,
     size, args, tune_params, answer=answer,
     observers=observers, lang="CUDA")
```

Kernel Tuner Example

```python
size = 100000000
n = numpy.int32(size)
a = numpy.random.randn(size)
b = numpy.random.randn(size)
c = numpy.zeros_like(b)

args = [n,
    TunablePrecision("float_type", a),
    TunablePrecision("float_type", b),
    TunablePrecision("float_type", c)]
answer = [None, None, None, a + b]

tune_params = dict()
tune_params["block_size_x"] = [32, 64, 128, 256, 512, 1024]
tune_params["float_type"] = ["float", "double", "half"]

observers = [AccuracyObserver("RMSE")]

results, env = tune_kernel("vector_add",kernel_string,
    size, args, tune_params, answer=answer,
    observers=observers, lang="CUDA")
```

Tunable and TunablePrecision

- The `TunablePrecision` wrapper tells Kernel Tuner that type of input/output arguments depends on a **tunable parameter**
- **Before** benchmarking, data **converted** to provided data types
- **During** benchmarking, kernel is passed **pointer** of correct data type
- [Advanced] The general `Tunable` object allows arbitrary conversions

Kernel Tuner Example

```python
size = 100000000
n = numpy.int32(size)
a = numpy.random.randn(size)
b = numpy.random.randn(size)
c = numpy.zeros_like(b)

args = [n,
    TunablePrecision("float_type", a),
    TunablePrecision("float_type", b),
    TunablePrecision("float_type", c)]

answer = [None, None, None, a + b]

tune_params = dict()
tune_params["block_size_x"] = [32, 64, 128, 256, 512, 1024]
tune_params["float_type"] = ["float", "double", "half"]

observers = [AccuracyObserver("RMSE")]

results, env = tune_kernel("vector_add", kernel_string,
    size, args, tune_params, answer=answer,
    observers=observers, lang="CUDA")
```

Kernel Tuner Example

```
size = 100000000
n = numpy.int32(size)
a = numpy.random.randn(size)
b = numpy.random.randn(size)
c = numpy.zeros_like(b)

 args = [n,
     TunablePrecision("float_type", a),
     TunablePrecision("float_type", b),
     TunablePrecision("float_type", c)]

 answer = [None, None, None, a + b]

 tune_params = dict()
 tune_params["block_size_x"] = [32, 64, 128, 256, 512, 1024]
 tune_params["float_type"] = ["float", "double", "half"]

 observers = [AccuracyObserver("RMSE")]

 results, env = tune_kernel("vector_add",kernel_string,
     size, args, tune_params, answer=answer,
     observers=observers, lang="CUDA")
```

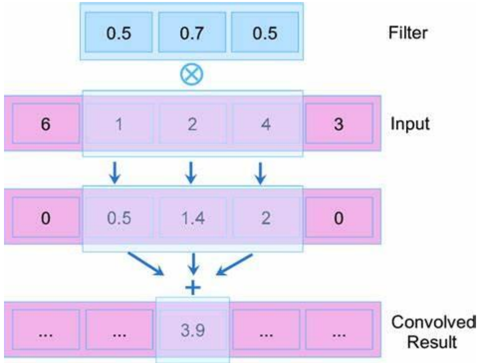
Error metrics

- The `AccuracyObserver` measures the error and adds a metric
- Supports 10+ well-known metrics
  - Root mean square error (`RMSE`)
  - Mean relative error (`rel`)
  - Mean absolute error (`abs`)
  - Maximum relative error (`max`)
  - ….
  - Custom metrics are also possible!
- The best error metric is **application-dependent**
- Compatible with other observers!

Hands-on

Second hands-on

- The second hands-on notebook is:
  - https://github.com/KernelTuner/kernel_tuner_tutorial/blob/master/hands-on/esiwace3/05_Mixed_precision_programming.ipynb



- The goal of this hands-on is to:
  - Tune a signal convolution kernel with mixed precision types
  - Experiment with the accuracy-performance trade-off

- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors

```
# The tunable types. Currently, the code is only tuned for double
tune_params = dict()
tune_params["OUTPUT_TYPE"] = ["double"]
tune_params["INPUT_TYPE"] = ["double"]
tune_params["FILTER_TYPE"] = ["double"]
tune_params["ACCUM_TYPE"] = ["double"]

# Other tunable parameters
tune_params["block_size_x"] = [128, 256]
tune_params["VECTOR_SIZE"] = [1, 2, 4]
tune_params["PREFETCH_INPUT"] = [0, 1]
tune_params["UNROLL_LOOP"] = [0, 1]
```

Optimizing GPU Core Clock Frequency

Measuring power consumption with NVML

NVML (the Nvidia Management Library) can observe GPU temperature, core and memory clocks, core voltage, and power

Advantages:

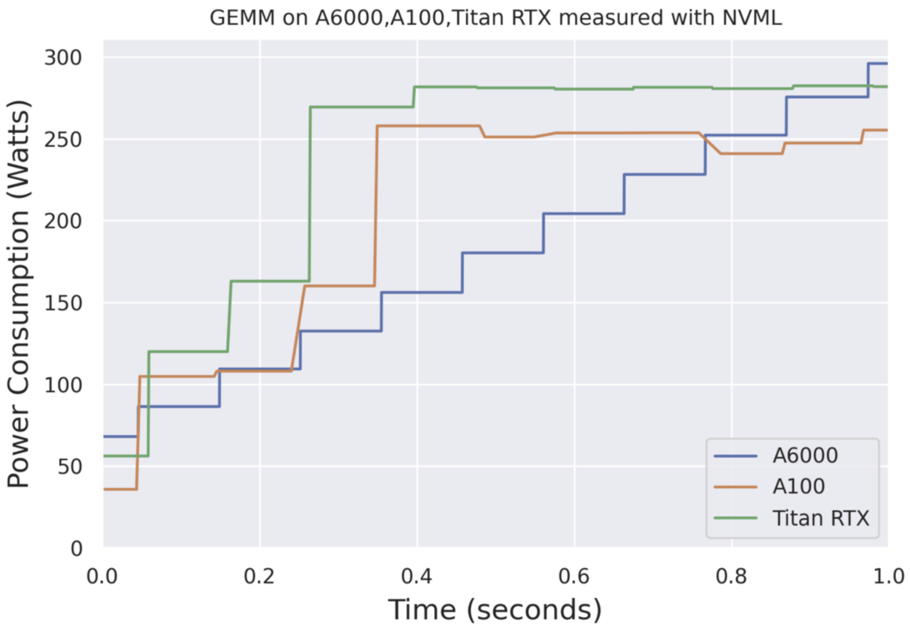
- Highly available

Disadvantages:

- Returns time-averaged power, not instantaneous power consumption
- Limited time resolution

Current solution:

- Measure power while continuously running the kernel for one second



Going Green: optimizing GPUs for energy efficiency through model-steered auto-tuning
Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
PMBS workshop at SC22 2022

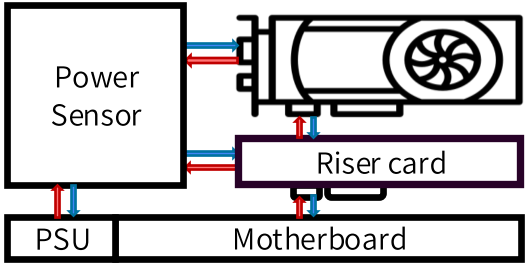
PowerSensor2

Pros:

- Instantaneous power readings
- Time resolution: 2.8 KHz
- Open source: https://gitlab.com/astron-misc/PowerSensor

Cons:

- Some assembly required
  - You need to build the hardware!



Supported in Kernel Tuner, using `PowerSensorObserver`

PowerSensor 2: A Fast Power Measurement Tool, John W. Romein and Bram Veenboer, International Symposium on Performance Analysis of Systems and Software (ISPASS 2018)

NVMLObserver - Nvidia Management Library (NVML)

Allows to measure several quantities during tuning:

- Power consumption, core frequency, core voltage, memory frequency, GPU temperature, and energy consumption

Provides an interface within Kernel Tuner to NVML:

- Enables new tunable parameters:
  - `nvml_pwr_limit`: try out different power limits
  - `nvml_gr_clock`: set the GPU core clock frequency
  - `nvml_mem_clock`: set the GPU memory clock frequency
  - Setting these requires root privileges

NVMLObserver example

```
...
tune_params["nvml_pwr_limit"] = [250, 225, 200, 175]

nvmlobserver = NVMLObserver(["nvml_energy", "temperature"])

metrics = OrderedDict()
metrics["GFLOPS/W"] = lambda p: (size/1e9) / p["nvml_energy"]

results, env = tune_kernel("vector_add", kernel_string, size, args,
                           tune_params, observers=[nvmlobserver],
                           metrics=metrics, iterations=32)
```

Setup GPU power limits and core and memory clock frequencies

- Kernel Tuner has helper functions to setup tunable parameters:

In `kernel_tuner.observers.nvml`:

- `get_nvml_pwr_limits(device, n=None, quiet=False):`
    - Device is the device ordinal as reported by `nvidia-smi`
    - n is the number of evenly-spaced values to tune
        - if unspecified returns values spaced 5 Watts apart
- `get_nvml_gr_clocks(device, n=None, quiet=False):`
    - n is the number of evenly-spaced values to tune
        - If unspecified, all supported core clocks are returned

Setup core clock frequencies for tuning

```
tune_params["block_size_x"] = [128, 256, 512, 1024]
...

tune_params.update(get_nvml_gr_clocks(0, 7)) # get 7 core frequencies supported on device 0

nvmlobserver = NVMLObserver(["nvml_energy", "temperature"])

metrics = OrderedDict()
metrics["GFLOPS/W"] = lambda p: (size/1e9) / p["nvml_energy"]

results, env = tune_kernel("vector_add", kernel_string, size, args,
                           tune_params, observers=[nvmlobserver],
                           metrics=metrics, iterations=32)
```

Power limit vs frequency tuning

Many tunable parameters affect compute performance and/or energy efficiency

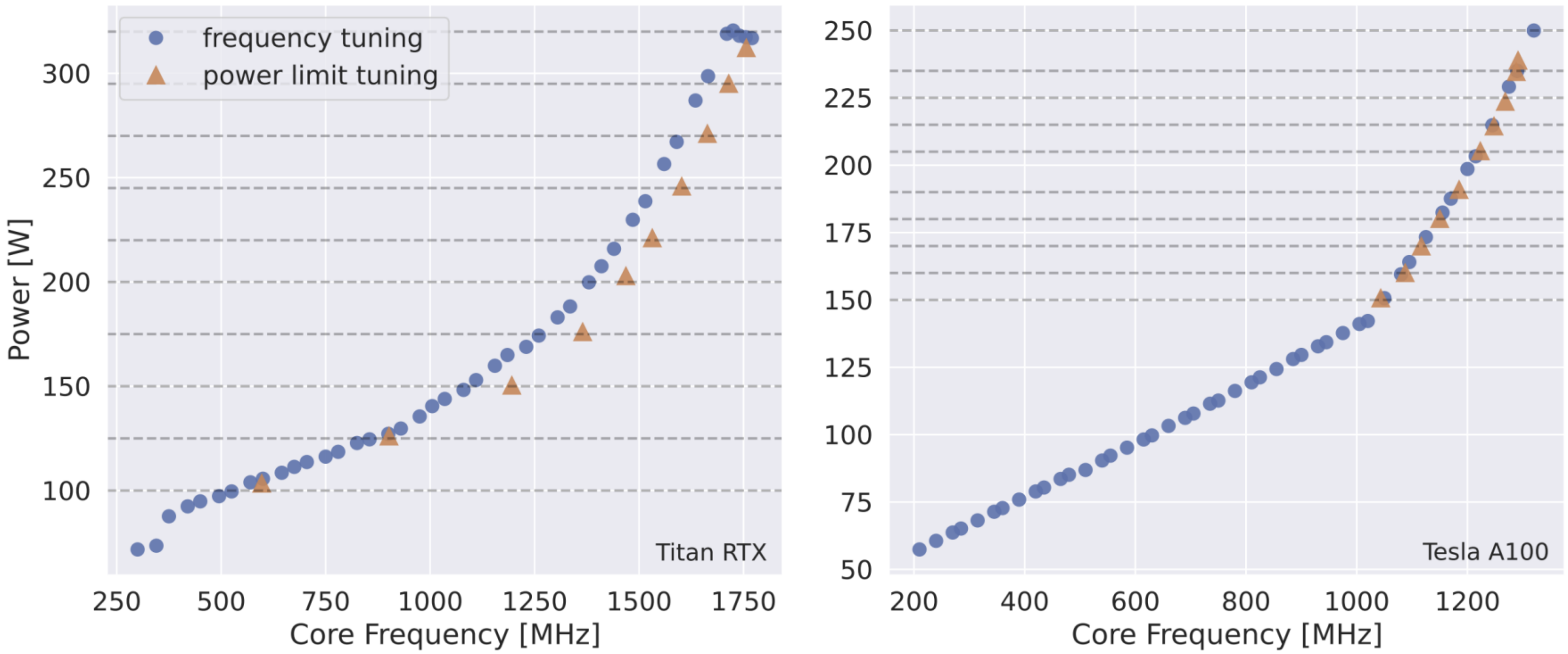
But we can also:

- Limit the GPU clock frequency, allow GPU to vary power consumption
- Limit the GPU power consumption, allow GPU to determine clock frequency

Both methods unfortunately require root privileges for the latest generations of Nvidia GPUs

Frequency-power relation

Tuning CLBlast GEMM using frequency or power limit tuning



Frequency tuning vs power capping

Advantages of power capping:

- Potentially more effective, GPU may also lower memory clock
- Reliable method in face of limited power

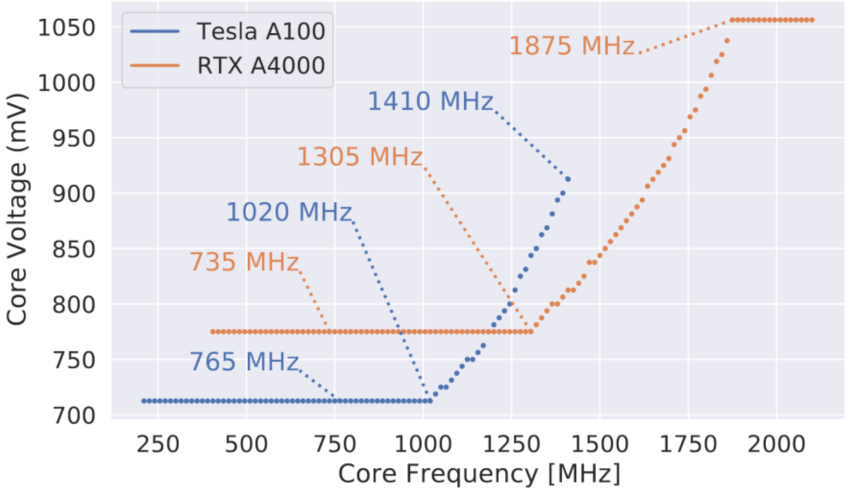
Advantages of frequency tuning:

- Especially on A100, frequency tuning enables a wider power range
- Fixing the clock frequency also improves measurement stability

Frequency Voltage relation

- GPUs rapidly ramp up voltage when clock frequency increases beyond a certain point
- This point appears to be a sweet spot in the trade-off between energy consumption and compute performance
- We call this point the 'ridge point'

Observation: base clock, ridge point, max



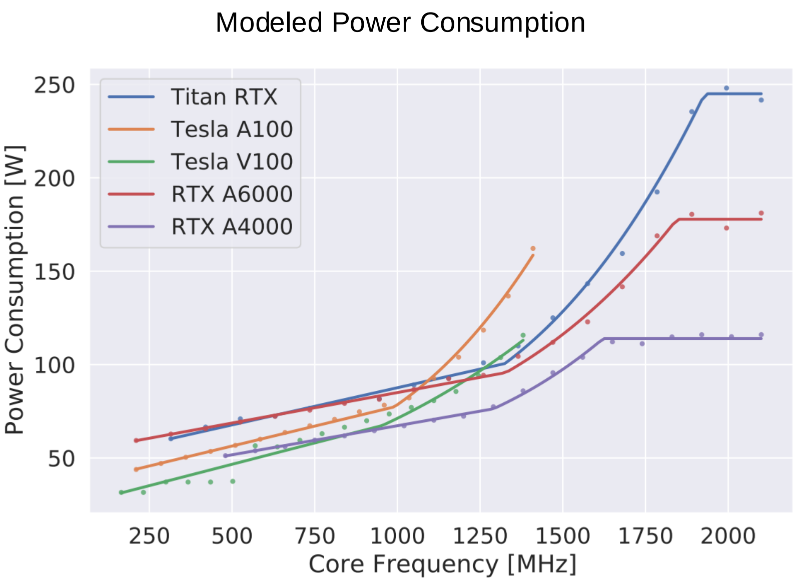
Going Green: optimizing GPUs for energy efficiency through model-steered auto-tuning
Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
PMBS workshop at SC22 2022

A simple power consumption model

- Not every GPU reports core voltages, but we can estimate the voltage using a simple power model
- When we fix all parameters and vary the clock frequency, we can approximate power consumption using:

$$P_{load} = \min(P_{max}, P_{idle} + \alpha * f * v^2)$$

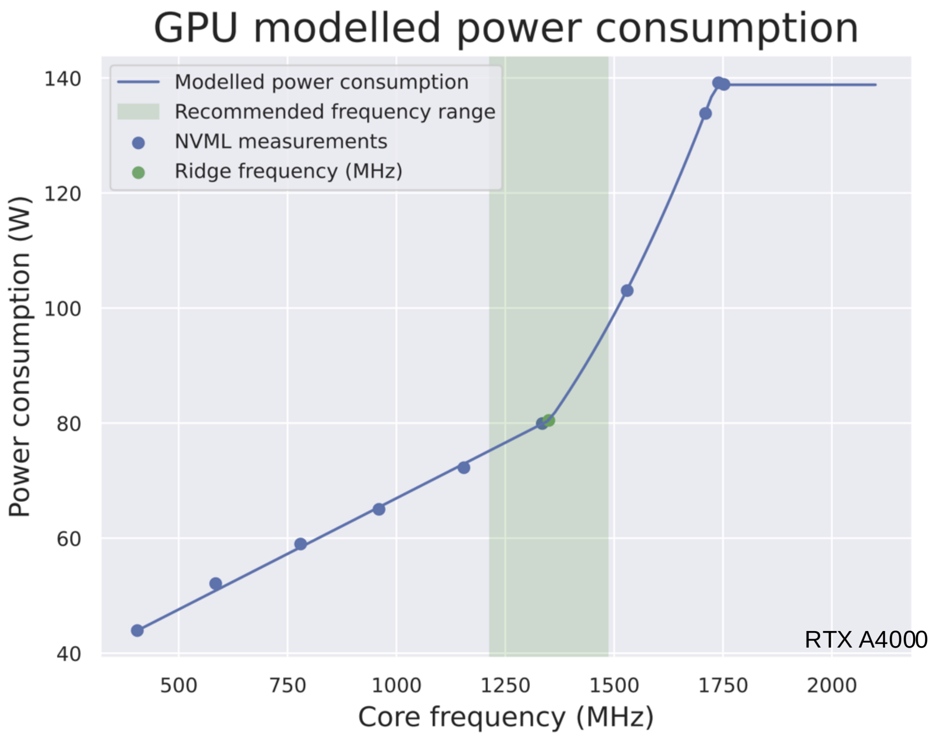
- And identify the GPUs 'ridge point' frequency in this way



Going Green: optimizing GPUs for energy efficiency through model-steered auto-tuning
Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
PMBS workshop at SC22 2022

Model-steered auto-tuning

- Use performance model to limit the frequency range for tuning
- Reduces the search space by ~80% on average



https://github.com/KernelTuner/kernel_tuner/blob/master/examples/cuda/going_green_performance_model.py

Custom tuning objectives

- By default, Kernel Tuner's optimization strategy minimizes `time`
- But there is also support for using a custom tuning `objective`
- The objective can be any observed quantity or user-defined metric

```
metrics["GFLOPS/W"] = lambda p: (size/1e9) / p["nvml_energy"]

results, env = tune_kernel("vector_add", kernel_string, size, args,
                           tune_params, observers=[nvmlobserver],
                           metrics=metrics, iterations=32,
                           objective="GFLOPS/W")
```

Hands-on

Third hands-on

- The third hands-on notebook is:
  - https://github.com/KernelTuner/kernel_tuner_tutorial/blob/master/hands-on/esiwace3/06_Energy_Efficient_Computing.ipynb

- The goal of this hands-on is to:
  - Tune a kernel to minimize the execution time or the energy consumption
  - Use an optimization strategy
  - Compare different energy optimization strategies

- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors

Closing Remarks

How do we create energy efficient GPU applications?

Three strategies for energy efficient GPU Computing:

1. Use for shorter amount of time → Optimize application performance
2. Minimize data movements → Lower/mixed precision techniques
3. Optimize device settings → Optimize clock frequency

Learning objectives

- Understand the energy footprint of computing
- Optimize applications for performance to reduce energy consumption
- Reduce data movement with mixed-precision techniques
- Tune GPU core frequencies to find the most energy-efficient configuration

Contributions are welcome!

- Contributions can come in many forms: tweets, blog posts, issues, pull requests
- Before making larger changes, please create an issue to discuss

- For the full contribution guide, please see:
  https://kerneltuner.github.io/kernel_tuner/stable/contributing.html

Related publications

- **Kernel Launcher: C++ library for creating optimal-performance portable CUDA applications**
  S. Heldens, B. van Werkhoven
  International Workshop on Automatic Performance Tuning (iWAPT2023) co-located with IPDPS 2023
- **Optimization Techniques for GPU Programming**
  Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri Bal
  ACM Computing surveys 2023
- **Going green: optimizing GPUs for energy efficiency through model-steered auto-tuning**
  Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
  International Workshop on Performance Modeling, Benchmarking and Simulation of High-Performance Computer Systems (PMBS) at Supercomputing (SC22) 2022
- **Bayesian Optimization for auto-tuning GPU kernels**
  F.J. Willemsen, R.V. van Nieuwpoort, B. van Werkhoven
  International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) at Supercomputing (SC21) 2021
- **Kernel Tuner: A search-optimizing GPU code auto-tuner**
  B. van Werkhoven
  Future Generation Computer Systems 2019

Interested in getting in touch?

Website: www.esiwace.eu

Twitter: https://twitter.com/esiwace

YouTube: https://www.youtube.com/@esiwace880

ESiWACE is on Zenodo, the Open Access repository for scientific results
https://zenodo.org/communities/esiwace

Place and date of meeting

Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) under grant agreement No 101093054.