

Heterogeneous computing with performance modelling

Performance modelling

Mirko Myllykoski
mirkom@cs.umu.se

Department of Computing Science and HPC2N
Umeå University

4-5. November 2020



Remark

One should always strive to measure everything multiple times and compute **averages** etc!

During this course, we are measuring only once for time and resource saving reasons (50 participants, 6 GPUs).

How do we measure performance?

Floprate (definition)

- ▶ The raw computing performance of a CPU or a GPU is usually measured in **Flops**. That is,

$$\text{Floprate} = \frac{\text{number of floating-point operations [Flop]}}{\text{time [s]}}.$$

- ▶ Usually the number of floating-point additions and multiplications the hardware can perform per second.
 - ▶ Additions and multiplications are usually faster (FMA).
 - ▶ Division and special functions are usually slower.

Floprate (theoretical peak floprate, double precision)

- ▶ A **theoretical peak floprate** can be calculated for each device.
- ▶ Quad-core Intel Skylake CPU:

~ 200 GFlops

- ▶ 14-core Intel Xeon Gold 6132 CPU:

~ 1200 GFlops

- ▶ Nvidia Tesla V100 GPU:

~ **7 000** GFlops

The Nvidia Tesla V100 GPU is **over 11 times faster** than the 14-core Intel Xeon CPU!

Floprate (single and half precision)

- ▶ The difference is even larger if we are willing to reduce the precision.
- ▶ Typical numbers (single precision):
 - ▶ Quad-core Intel Skylake CPU: ~ 400 GFlops
 - ▶ 14-core Intel Xeon Gold 6132 CPU: $\sim 2\,400$ GFlops
 - ▶ Nvidia Tesla V100 GPU: $\sim 14\,000$ GFlops
- ▶ Typical numbers (half precision):
 - ▶ Quad-core Intel Skylake CPU: $\sim \text{—}$ GFlops
 - ▶ $2 \times$ Intel Xeon Gold 6132 CPU: $\sim \text{—}$ GFlops
 - ▶ Nvidia Tesla V100 GPU: $\sim \mathbf{112\,000}$ GFlops

The Nvidia Tesla V100 GPU is **over 90 times faster** than the 14-core Intel Xeon CPU!

AX example

- ▶ Let's perform a small experiments:

$$\alpha \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^n,$$

$$\mathbf{x} \leftarrow \alpha \mathbf{x}$$

- ▶ The total number of flops is n . Total number of bytes moved is $16n$.
- ▶ CPU code would look like this:

```
void ax(int n, double alpha, double *x)
{
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
        x[i] = alpha * x[i];
}
```

- ▶ GPU code would look like this:

```
__global__ void ax_kernel(int n, double alpha, double *x)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    int thread_count = gridDim.x * blockDim.x;
    for (int i = thread_id; i < n; i += thread_count)
        x[i] = alpha * x[i];
}
```

AX example (actual performance)

- ▶ Quad-core Intel Skylake CPU (~ 200 GFlops):
\$ OMP_NUM_THREADS=4 ./ax.cpu 500E6
Time = 0.291716 s
- ▶ 14-core Intel Xeon Gold 6132 CPU ($\sim 1\,200$ GFlops):
\$ OMP_NUM_THREADS=14 srun/ax.cpu 500E6
Time = 0.087790 s
- ▶ Nvidia Tesla V100 GPU ($\sim 7\,000$ GFlops):
\$ srun/ax.cuda 500E6
Time = 0.010582 s

AX example (actual speedup)

The V100 is over 8 times faster than
the Xeon but ...



AX example (actual floprate)

- ▶ Quad-core Intel Skylake CPU (~ 200 GFlops):
\$ OMP_NUM_THREADS=4 ./ax.cpu 500E6
Time = 0.291716 s
Floprate = 2 GFlops
- ▶ 14-core Intel Xeon Gold 6132 CPU (~ 1200 GFlops):
\$ OMP_NUM_THREADS=14 srun/ax.cpu 500E6
Time = 0.087790 s
Floprate = 6 GFlops
- ▶ Nvidia Tesla V100 GPU ($\sim 7\ 000$ GFlops):
\$ srun/ax.cuda 500E6
Time = 0.010582 s
Floprate = 47 GFlops

The V100 is over 8 times faster than the Xeon but **we are using less than 1% of the peak floprate!**

Why?

What else could effect the performance?

Memory throughput (definition)

- ▶ The memory performance of a CPU or a GPU is usually measured in terms of **memory throughput**. That is,

$$\text{Throughput} = \frac{\text{number of bytes moved [Byte]}}{\text{time [s]}}.$$

- ▶ Usually the bandwidth is measured between the CPU cores and the main memory; or the CUDA cores and the global memory.

Memory throughput (theoretical memory bandwidth)

- ▶ A **theoretical memory bandwidth** can be calculated for each device.
- ▶ Quad-core Intel Skylake CPU:

~ 35 GB/s

- ▶ 14-core Intel Xeon Gold 6132 CPU:

~ 100 GB/s

- ▶ Nvidia Tesla V100 GPU:

~ **900** GB/s

AX example (actual memory throughput)

- ▶ Quad-core Intel Skylake CPU (~ 35 GB/s):

```
$ OMP_NUM_THREADS=4 ./ax.cpu 500E6
```

```
Time = 0.291716 s
```

```
Floprate = 2 GFlops
```

```
Memory throughput = 27 GB/s
```

- ▶ 14-core Intel Xeon Gold 6132 CPU (~ 100 GB/s):

```
$ OMP_NUM_THREADS=14 srun .... ./ax.cpu 500E6
```

```
Time = 0.087790 s
```

```
Floprate = 6 GFlops
```

```
Memory throughput = 91 GB/s
```

- ▶ Nvidia Tesla V100 GPU (\sim **900** GB/s):

```
$ srun .... ./ax.cuda 500E6
```

```
Time = 0.010582 s
```

```
Floprate = 47 GFlops
```

```
Memory throughput = 756 GB/s
```

We are using between 77% and **91%**
of the memory bandwidth!

AX example (profiler)

- ▶ We can use Nvidia's `nv-nsight-cu-cli` profiling tool to analyze the situation:

```
$ srun .... nv-nsight-cu-cli ./ax.cuda 500E6
....
```

```
-----
Memory Frequency  cycle/usecond          875,84
SOL FB              %              84,58
Elapsed Cycles      cycle      13 181 137
SM Frequency        cycle/nsecond        1,25
Memory [%]          %              84,58
Duration            msecond          10,55
SOL L2              %              31,21
SM Active Cycles    cycle      12 837 852,30
SM [%]              %              3,34
SOL TEX             %              15,21
```

AX example (profiler)

- ▶ The relevant fields are the following:

SOL FB	%	84,58
Memory [%]	%	84,58
SM [%]	%	3,34

- ▶ SOL FB is related to global memory throughput and Memory is related to the occupancy rate of the memory subsystem.
- ▶ SM is related to the occupancy rate of the compute resources.
- ▶ Conclusion: **The CUDA cores are idling but the memory bus is busy.**

GEMM example

- ▶ Lets perform a second experiments:

$$\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$$

$$\mathbf{C} \leftarrow \mathbf{AB}, \mathbf{C} \in \mathbb{R}^{n \times n}$$

- ▶ A naive CPU code would look like this:

```
void gemm(int n, int ldA, int ldB, int ldC, double *A, double *B, double *C)
{
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            double dot = 0.0;
            for (int k = 0; k < n; k++)
                dot += A[k*ldA+j] * B[i*ldB+k];
            C[i*ldC+j] = dot;
        }
    }
}
```

- ▶ Total number of flops is $2n^3$. Total number of bytes transferred is $16n^3 + 8n^2$ (this does not hold in practice).

GEMM example (actual floprate)

- ▶ An optimized BLAS library was used to generate these results.
- ▶ Quad-core Intel Skylake CPU (~ 200 GFlops):
\$ OMP_NUM_THREADS=4 ./gemm.cpu 5000
Runtime was 1.422 s.
Floprate was 176 GFlops.
- ▶ 14-core Intel Xeon Gold 6132 CPU (~ 1200 GFlops)¹:
\$ OMP_NUM_THREADS=14 srun/gemm.cpu 5000
Runtime was 0.469 s.
Floprate was 533 GFlops
- ▶ Nvidia Tesla V100 GPU (~ 7000 GFlops):
\$ srun/gemm.cuda 5000
Runtime was 0.041 s.
Floprate was 6077 GFlops

¹I am using OpenBLAS (fosscuda). MKL would give better performance.

GEMM example (actual floprate)

We are using between 46% and **88%**
of the peak floprate!

GEMM example (profiler)

- ▶ Let's see what the profiler says:

```
$ srun .... nv-nsight-cu-cli ./gemm.cuda 5000
```

```
....
```

```
-----  
Memory Frequency  cycle/usecond           878,08  
SOL FB                %                36,45  
Elapsed Cycles      cycle           51 458 926  
SM Frequency        cycle/nsecond          1,25  
Memory [%]          %                37,18  
Duration            msecond           41,06  
SOL L2              %                23,69  
SM Active Cycles   cycle           50 755 341,94  
SM [%]              %                98,31  
SOL TEX             %                37,68
```

GEMM example (profiler)

- ▶ The relevant fields:

SOL FB	%	36,45
Memory [%]	%	37,18
SM [%]	%	98,31

- ▶ Conclusion: **The CUDA cores are busy but the memory bus is party idle.**

The two codes behave very differently...

Can we predict the behavior in advance?

Arithmetical intensity

- ▶ From now on, let's call
 - ▶ the former type of kernels (ax) **memory bound** and
 - ▶ the latter type of kernels (gemm) **compute bound**.
- ▶ That is,
 - ▶ the performance of a memory-bound code is limited by the available memory bandwidth and
 - ▶ the performance of a compute-bound code is limited by the instruction throughput.

Arithmetical intensity (definition)

- ▶ How do we know which kernels are memory bound and which are compute bound?
- ▶ We begin to answer this question by defining **arithmetical intensity**:

$$\text{Arithmetical intensity} = \frac{\text{number of floating-point operations [Flop]}}{\text{number of bytes moved [Byte]}}.$$

Arithmetical intensity (examples)

- ▶ Double precision AX has the arithmetical intensity of

$$\text{Arithmetical intensity}_{AX, \text{double}} = \frac{1 \text{ Flop}}{2 \cdot 8 \text{ Byte}} = \frac{1}{16} \text{ Flop/Byte.}$$

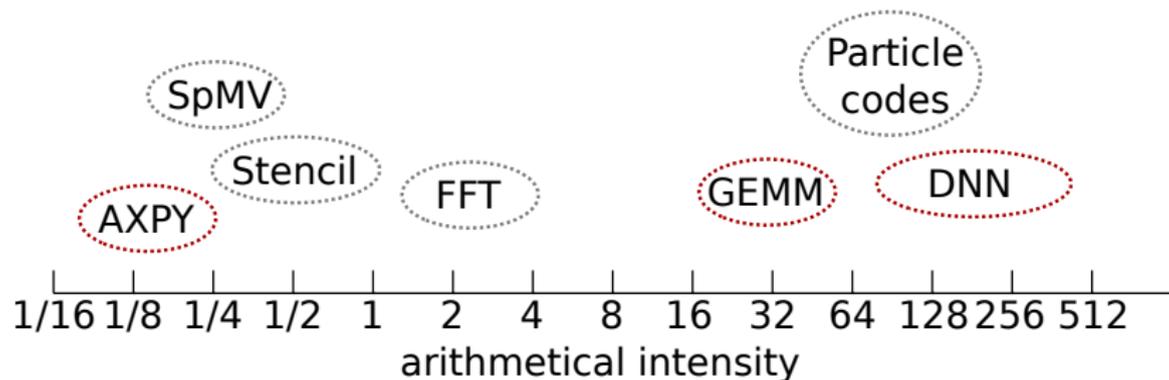
- ▶ Single precision AX has the arithmetical intensity of

$$\text{Arithmetical intensity}_{AX, \text{single}} = \frac{1 \text{ Flop}}{2 \cdot 4 \text{ Byte}} = \frac{1}{8} \text{ Flop/Byte.}$$

- ▶ Well-implemented double-precision GEMM has the arithmetical intensity of

$$\text{Arithmetical intensity}_{GEMM, \text{double}} = \sim 32 \text{ Flop/Byte}$$

Arithmetical intensity (more examples)



Arithmetical intensity (deep neural networks)

► Half-precision numbers from Nvidia:

Operation	Arithmetical intensity
Linear layer (4096 outputs, 1024 inputs, batch size 512)	315 Flop/Byte
Linear layer (4096 outputs, 1024 inputs, batch size 1)	1 Flop/Byte
Max pooling with 3x3 window and unit stride	2.25 Flop/Byte
ReLU activation	0.25 Flop/Byte
Layer normalization	< 10 Flop/Byte

Arithmetical intensity (what can we do with it?)

- ▶ Let's (loosely) define the total amount of **work** as

$$\text{Work} = \text{Flops} + \text{Transfers}.$$

- ▶ If we assume that floating-point and memory operations are more or less evenly distributed throughout the code, we can estimate the **execution time** with

$$\text{Time} = \max \left\{ \frac{\text{Flops}}{\text{Peak Floprate}}, \frac{\text{Transfers}}{\text{Bandwidth}} \right\}.$$

Arithmetical intensity (what can we do with it?)

- ▶ Now, since Performance = Flops / Time, we have

$$\begin{aligned} \text{Performance} &= \frac{\text{Flops}}{\max \left\{ \frac{\text{Flops}}{\text{Peak Floprate}}, \frac{\text{Transfers}}{\text{Bandwidth}} \right\}} \\ &= \min \left\{ \text{Peak Floprate}, \frac{\text{Flops}}{\text{Transfers}} \times \text{Bandwidth} \right\}. \end{aligned}$$

- ▶ Note that the right term is simply **the arithmetical intensity multiplied by the memory bandwidth!**

Arithmetical intensity (optimal intensity)

- ▶ An **optimal arithmetical intensity** can be calculated for each device:

$$\text{Optimal intensity} = \frac{\text{theoretical peak floprate}}{\text{theoretical memory bandwidth}}.$$

- ▶ In that case, we have

$$\begin{aligned} \text{Performance} &= \min \left\{ \text{Peak Floprate}, \frac{\text{Peak Floprate}}{\text{Bandwidth}} \times \text{Bandwidth} \right\} \\ &= \text{Peak Floprate}. \end{aligned}$$

Arithmetical intensity (optimal intensity)

- ▶ If the arithmetical intensity is **smaller than** the optimal intensity, the kernel is **memory bound** and we have

$$\text{Performance} = \frac{\text{Flops}}{\text{Transfers}} \times \text{Bandwidth} \\ \leq \text{Peak Floprate.}$$

- ▶ If the arithmetical intensity is **larger than** the optimal intensity, the kernel is **compute bound** and we have

$$\text{Performance} = \text{Peak Floprate.}$$

Arithmetical intensity (optimal intensity, double precision)

- ▶ Quad-core Intel Skylake CPU:

~ 5.7 Flop/Byte

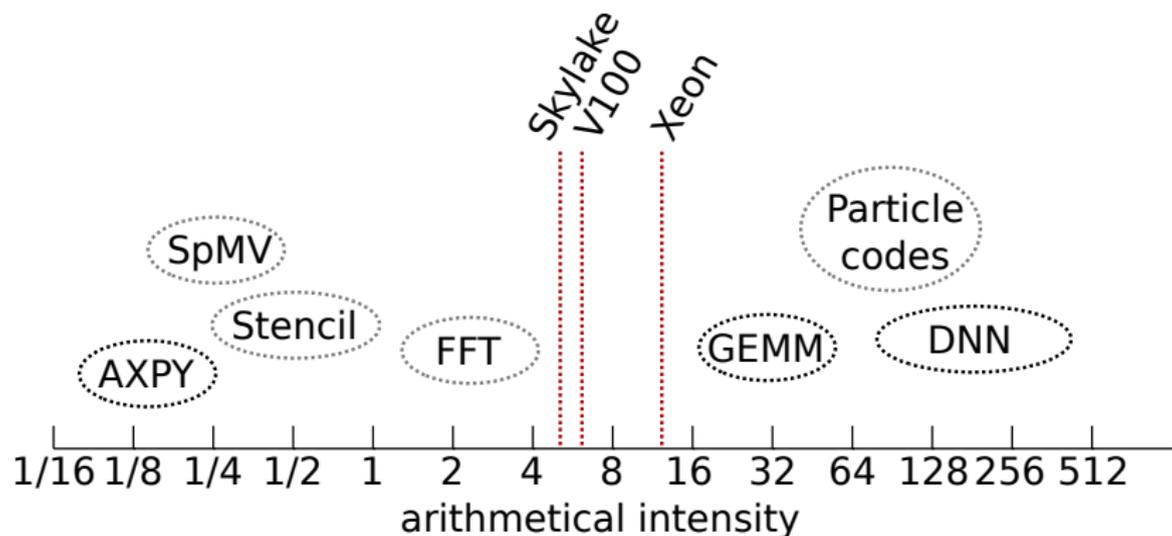
- ▶ 14-core Intel Xeon Gold 6132 CPU:

~ 12 Flop/Byte

- ▶ Nvidia Tesla V100 GPU:

~ 7.7 Flop/Byte

Arithmetical intensity (optimal intensity, double precision)



Arithmetical intensity (optimal intensity, single precision)

- ▶ Quad-core Intel Skylake CPU:

~ 11.4 Flop/Byte

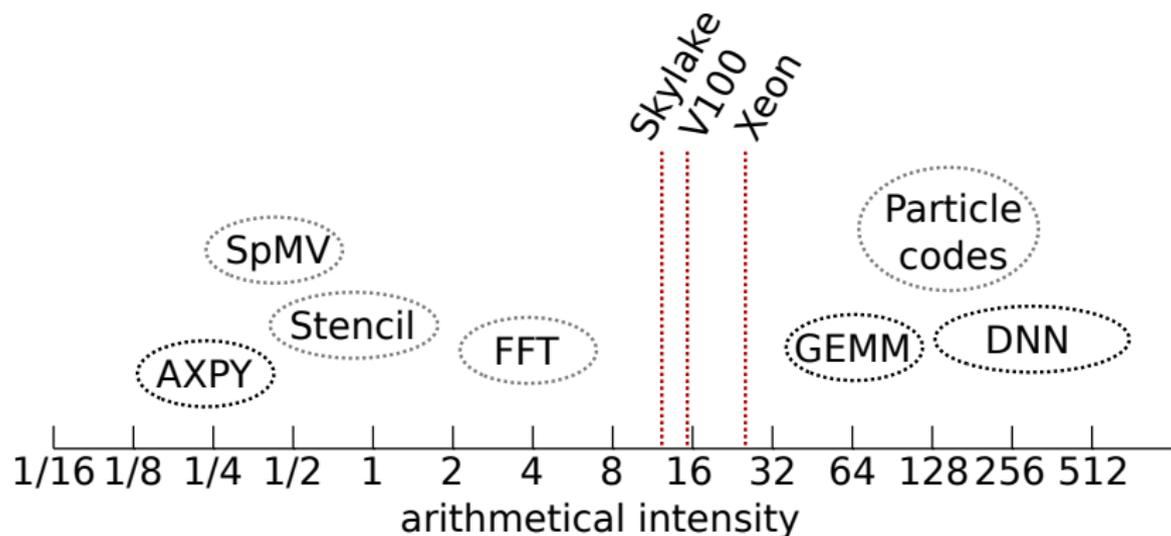
- ▶ 14-core Intel Xeon Gold 6132 CPU:

~ 24 Flop/Byte

- ▶ Nvidia Tesla V100 GPU:

~ 15.6 Flop/Byte

Arithmetical intensity (optimal intensity, double precision)



Arithmetical intensity (optimal intensity, half precision)

- ▶ Quad-core Intel Skylake CPU:

— Flop/Byte

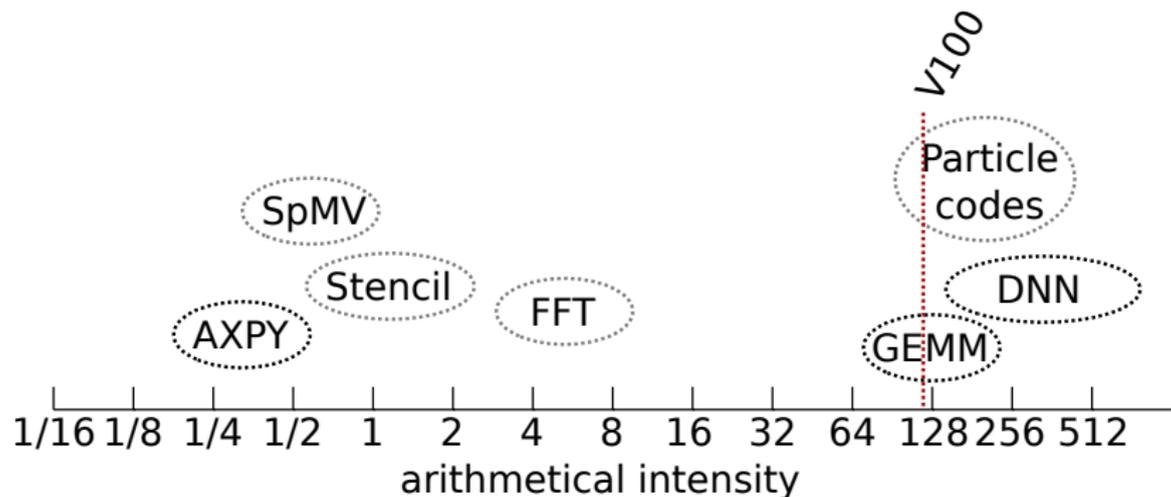
- ▶ 14-core Intel Xeon Gold 6132 CPU:

— Flop/Byte

- ▶ Nvidia Tesla V100 GPU:

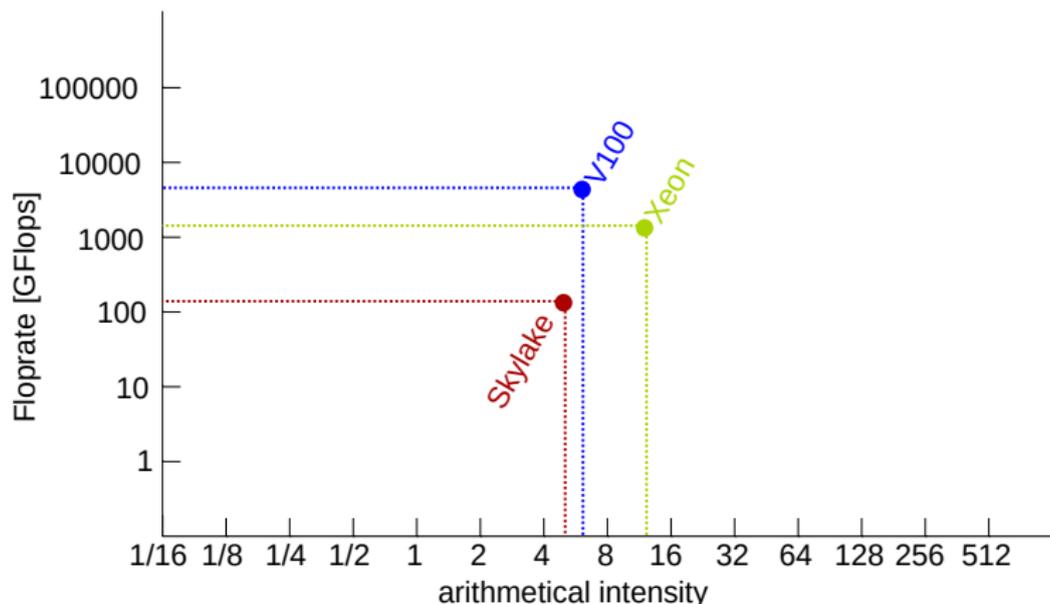
~ **124** Flop/Byte

Arithmetical intensity (optimal intensity, half precision)



Roofline model

- ▶ We could plot both the **peak floprate** and the **optimal arithmetical intensity** to the same figure:

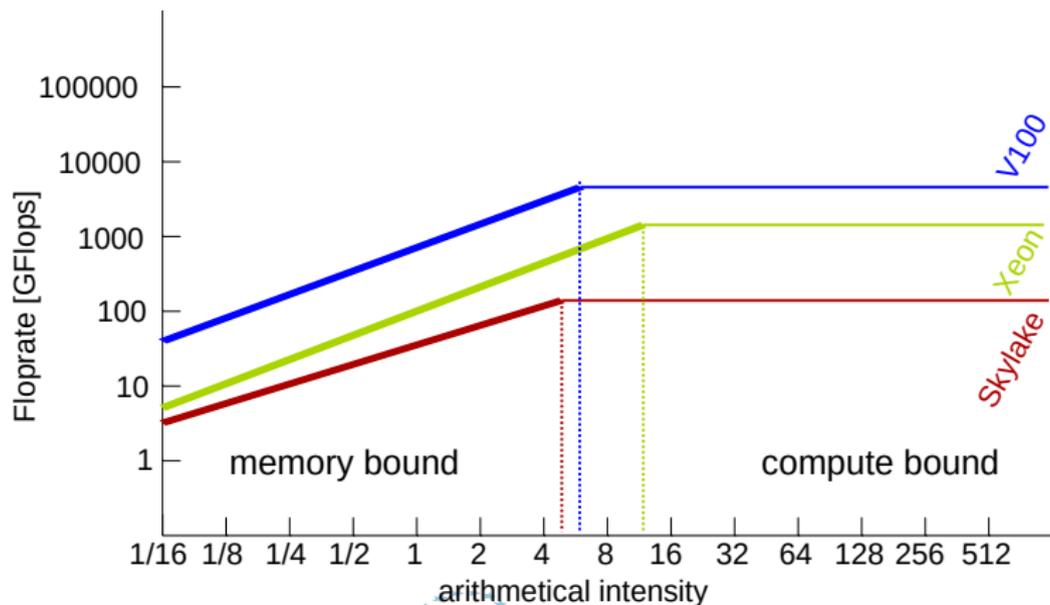


Roofline model (model)

- By including the function

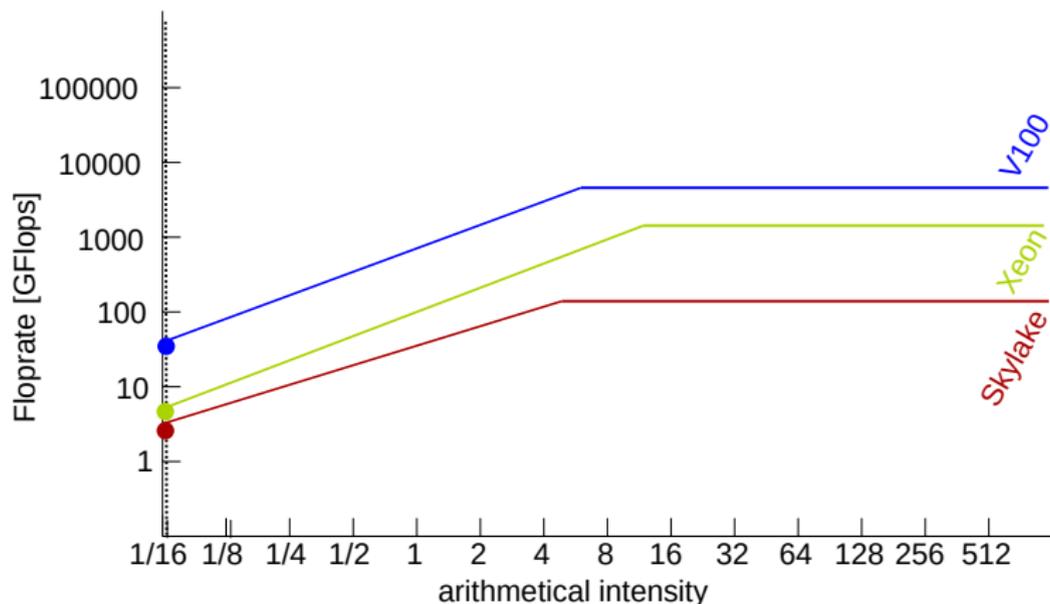
$$Performance = \min \left\{ \text{Peak Floprate}, \frac{\text{Flops}}{\text{Transfers}} \times \text{Bandwidth} \right\}$$

to the plot, we get the following **"roofline"**:



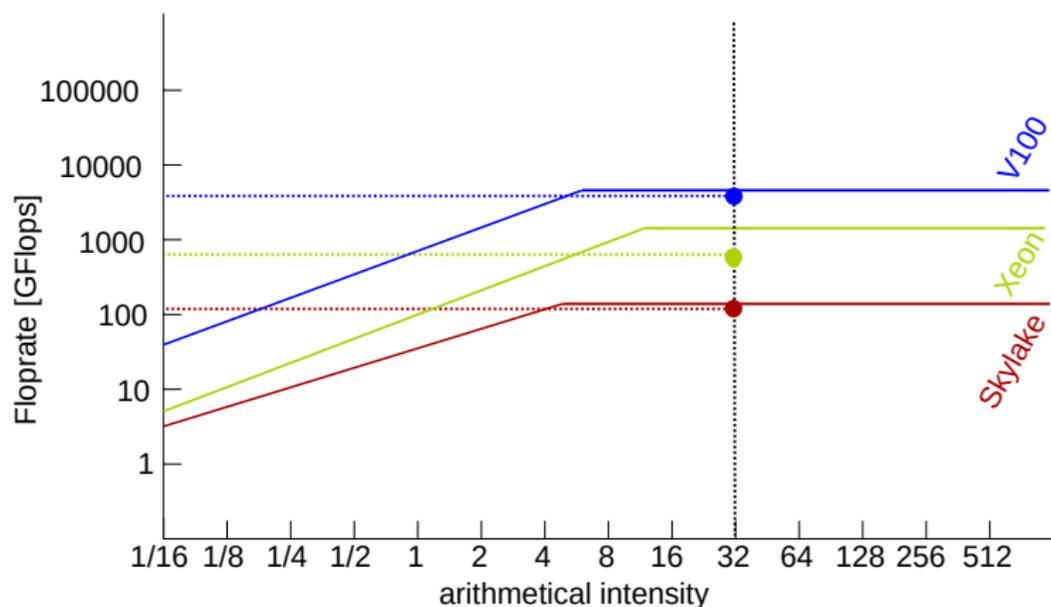
Roofline model (AX kernel)

- By including "Arithmetical intensity_{AX,double}" to the figure, we see that the measured floprate is actually quite close to the value predicted by the model:



Roofline model (GEMM kernel)

- ▶ The same applies to "Arithmetical intensity_{GEMM,double}":



Arithmetical intensity (caches and shared memory)

- ▶ When calculated naively, the double precision GEMM has the arithmetical intensity of

$$\begin{aligned}\text{Arithmetical intensity}_{\text{GEMM,double}} &= \frac{2n^3}{16n^3 + 8n^2} \text{ Flop/Byte} \\ &= \sim \frac{1}{8} \text{ Flop/Byte}\end{aligned}$$

- ▶ Why is the real number

$$\text{Arithmetical intensity}_{\text{GEMM,double}} = \sim 32 \text{ Flop/Byte?}$$

Arithmetical intensity (caches and shared memory)

- ▶ When implemented naively, we compute each entry separately:

$$\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}, (\mathbf{AB})_{i,j} = \sum_{k=1}^n a_{ik} b_{kj} \quad \left(\frac{n^2 \cdot 2n}{8n^2(2n+1)} \text{ Flop/Byte} \right)$$

- ▶ However, we can also do the following:

$$\left(\begin{bmatrix} \mathbf{A}_{11} & \dots & \mathbf{A}_{1m} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{m1} & \dots & \mathbf{A}_{mm} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{11} & \dots & \mathbf{B}_{1m} \\ \vdots & \ddots & \vdots \\ \mathbf{B}_{m1} & \dots & \mathbf{B}_{mm} \end{bmatrix} \right)_{i,j} = \sum_{k=1}^m \mathbf{A}_{ik} \mathbf{B}_{kj}$$

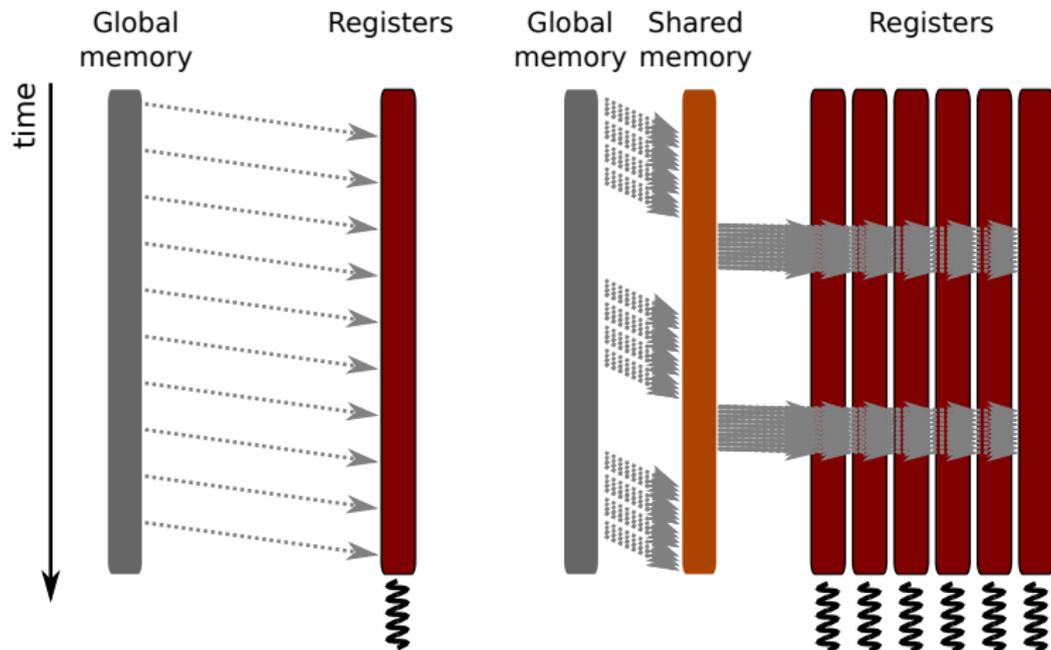
Arithmetical intensity (caches and shared memory)

- ▶ If the blocks \mathbf{A}_{ik} and \mathbf{B}_{kj} are small enough, they can be fitted into CPU caches or SMP's shared memory.
- ▶ Each block is loaded only once and then **shared among the thread block!**
- ▶ The matrix-matrix multiplication $\mathbf{A}_{ik}\mathbf{B}_{kj}$ can therefore be performed with minimal global memory communications.
- ▶ Only the global memory transfers are counted in the analysis
 - ⇒ the denominator decreases
 - ⇒ the arithmetical intensity increases
 - ⇒ higher performance on modern CPUs and GPUs.

Arithmetical intensity (caches and shared memory)

$$\sum_{k=1}^n a_{ik} b_{kj}$$

$$\sum_{k=1}^m \mathbf{A}_{ik} \mathbf{B}_{kj}$$



PCI-E bandwidth (data in global memory)

- ▶ In the earlier example, the necessary **data already resided in the global memory** when the timer was started:

```
struct timespec ts_start;  
clock_gettime(CLOCK_MONOTONIC, &ts_start);  
  
dim3 threads = 256;  
dim3 blocks = max(1, min(256, n/threads.x));  
ax_kernel<<<blocks, threads>>>(n, alpha, d_y);  
  
cudaDeviceSynchronize();  
  
struct timespec ts_stop;  
clock_gettime(CLOCK_MONOTONIC, &ts_stop);
```

- ▶ Outcome:

```
$ srun .... ./ax.cuda 500E6  
Time = 0.010582 s  
Floprate = 47 GFlops  
Memory throughput = 756 GB/s
```

PCI-E bandwidth (data in host memory)

▶ Let's change that:

```
struct timespec ts_start;
clock_gettime(CLOCK_MONOTONIC, &ts_start);

cudaMemcpy(d_y, y, n*sizeof(double), cudaMemcpyHostToDevice);

dim3 threads = 256;
dim3 blocks = max(1, min(256, n/threads.x));
ax_kernel<<<blocks, threads>>>(n, alpha, d_y);

cudaMemcpy(y, d_y, n*sizeof(double), cudaMemcpyDeviceToHost);

struct timespec ts_stop;
clock_gettime(CLOCK_MONOTONIC, &ts_stop);
```

▶ Outcome:

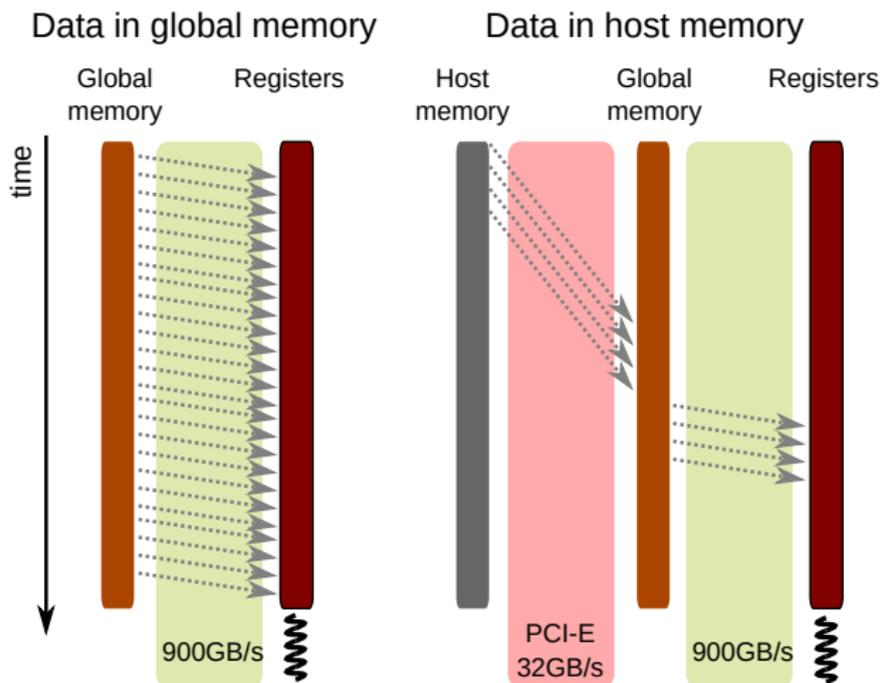
Time = 1.652748 s

Floprate = 0.3 GFlops

Memory throughput = 5 GB/s

PCI-E bandwidth (bandwidth)

- ▶ We must remember that the data must be transferred over the PCI-E bus:

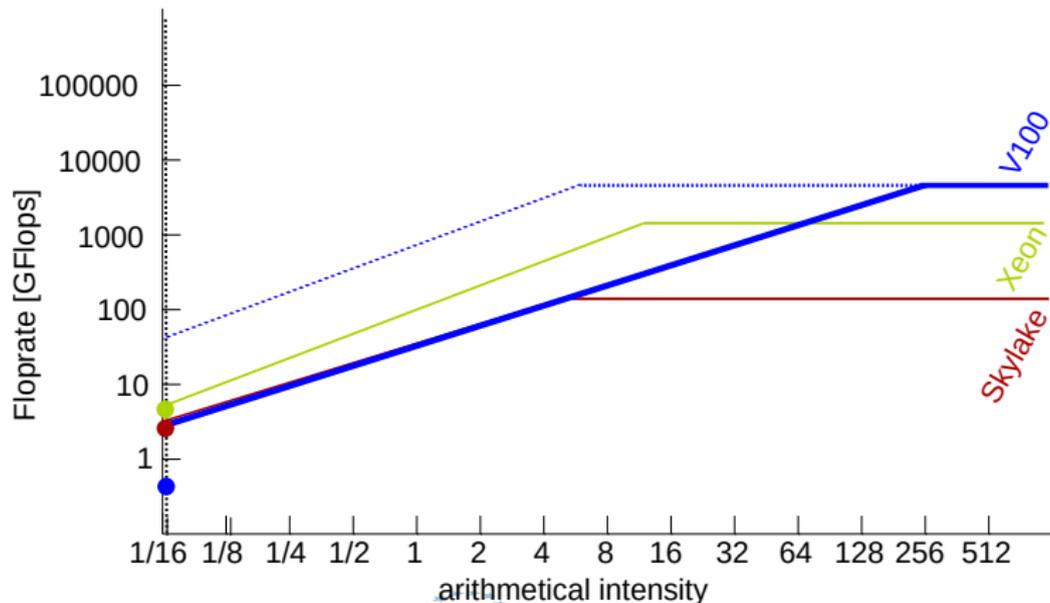


PCI-E bandwidth (roofline model)

- ▶ We must recalibrate the model for PCI-E 3.0:

$$Performance = \min \left\{ \text{Peak Floprate}, \frac{\text{Flops}}{\text{Transfers}} \times 32 \text{ BG/s.} \right\}$$

- ▶ The roofline plot is going to look very different:



Hands-ons

- ▶ Materials: https://git.cs.umu.se/mirkom/gpu_course/
- ▶ Two hands-ons under `hands-ons/3.modelling`:
 - 1.[compare](#) Analyze memory-bound and compute-bound codes.
 - 2.[profiling](#) Analyze and profile Wednesday's hands-ons.
- ▶ Solutions can be found under `solutions/3.modelling`.