Introduction to GPU programming: When and how to use GPU-acceleration?

Where is my performance?

Mirko Myllykoski
mirkom@cs.umu.se

Department of Computing Science / HPC2N
Umeå University

5 November 2019

UMEÅ UNIVERSITY

HPC2N

Performance

How do we measure performance?

Floprate (definition)

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

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

Floprate (definition)

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

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

- Usually the number of 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.

Floprate (theoretical peak floprate, double precision)

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

$\sim$ 200 GFlops

Floprate (theoretical peak floprate, double precision)

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

$\sim$ 200 GFlops

- 14-core Intel Xeon Gold 6132 CPU:

$\sim$ 1200 GFlops

Floprate (theoretical peak floprate, double precision)

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

$\sim$ 200 GFlops

- 14-core Intel Xeon Gold 6132 CPU:

$\sim$ 1200 GFlops

- Nvidia Tesla V100 GPU:

$\sim$ **7 000** GFlops

Floprate (theoretical speedup)

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.

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: $\sim$ 400 GFlops
  - 14-core Intel Xeon Gold 6132 CPU: $\sim$ 2 400 GFlops
  - Nvidia Tesla V100 GPU: $\sim$ 14 000 GFlops

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: $\sim$ 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: $\sim$ — GFlops
  - 2 $\times$ Intel Xeon Gold 6132 CPU: $\sim$ — GFlops
  - Nvidia Tesla V100 GPU: $\sim$ **112 000** GFlops

Floprate (single and half precision, theoretical speedup)

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

AXPY example (CPU)

- Lets perform a small experiments:

$$\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$$
$$\boldsymbol{y} \leftarrow 2\boldsymbol{x} + \boldsymbol{y}$$

AXPY example (CPU)

- Lets perform a small experiments:

$$\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$$

$$\boldsymbol{y} \leftarrow 2\boldsymbol{x} + \boldsymbol{y}$$

- CPU code would looks like this:

```
double *x = malloc(n*sizeof(double));
double *y = malloc(n*sizeof(double));

for (int i = 0; i < n; i++) {
    x[i] = 2.0 * rand()/RAND_MAX - 1.0;
    y[i] = 2.0 * rand()/RAND_MAX - 1.0;
}

// compute y <- 2 * x + y (level 1 BLAS routine)
cblas_daxpy(n, 2.0, x, 1, y, 1);
```

AXPY example (CPU)

- Lets perform a small experiments:

$$\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$$

$$\boldsymbol{y} \leftarrow 2\boldsymbol{x} + \boldsymbol{y}$$

- CPU code would looks like this:

```
double *x = malloc(n*sizeof(double));
double *y = malloc(n*sizeof(double));

for (int i = 0; i < n; i++) {
    x[i] = 2.0 * rand()/RAND_MAX - 1.0;
    y[i] = 2.0 * rand()/RAND_MAX - 1.0;
}

// compute y <- 2 * x + y (level 1 BLAS routine)
cblas_daxpy(n, 2.0, x, 1, y, 1);
```

- The total number of flops is $2n$.

AXPY example (CUDA)

- CUDA code would looks like this:

```
// allocate managed memory
double *x, *y;
cudaMallocManaged(&x, n*sizeof(double));
cudaMallocManaged(&y, n*sizeof(double));

for (int i = 0; i < n; i++) {
    x[i] = 2.0 * rand()/RAND_MAX - 1.0;
    y[i] = 2.0 * rand()/RAND_MAX - 1.0;
}

// prefetch data to GPU memory
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(x, n*sizeof(double), device, NULL);
cudaMemPrefetchAsync(y, n*sizeof(double), device, NULL);
cudaDeviceSynchronize();

// initialize cuBLAS
cublasHandle_t handle;
cublasCreate(&handle);

// compute y <- 2 * x + y (level 1 BLAS routine)
double alpha = 2.0;
cublasDaxpy(handle, n, &alpha, x, 1, y, 1);
```

AXPY example (actual performance)

- Quad-core Intel Skylake CPU ($\sim$ 200 GFlops):

```
$ ./axpy.cpu 500E6
Runtime was 0.484 s.
```

AXPY example (actual performance)

- Quad-core Intel Skylake CPU ($\sim$ 200 GFlops):

```
$ ./axpy.cpu 500E6
Runtime was 0.484 s.
```

- 14-core Intel Xeon Gold 6132 CPU ($\sim$ 1 200 GFlops):

```
$ srun --gres=gpu:v100:1,gpuexcl ... ./axpy.cpu 500E6
Runtime was 0.184 s.
```

AXPY example (actual performance)

- Quad-core Intel Skylake CPU ($\sim$ 200 GFlops):

```
$ ./axpy.cpu 500E6
Runtime was 0.484 s.
```

- 14-core Intel Xeon Gold 6132 CPU ($\sim$ 1 200 GFlops):

```
$ srun --gres=gpu:v100:1,gpuexcl ... ./axpy.cpu 500E6
Runtime was 0.184 s.
```

- Nvidia Tesla V100 GPU ($\sim$ **7 000** GFlops):

```
$ srun --gres=gpu:v100:1,gpuexcl ... ./axpy.cuda 500E6
Runtime was 0.014 s.
```

AXPY example (actual speedup)

The V100 is over 13 times faster than the Xeon but ...

AXPY example (actual floprate)

- Quad-core Intel Skylake CPU ($\sim$ 200 GFlops):

```
$ ./axpy.cpu 500E6
Runtime was 0.484 s.
Floprate was 2 GFlops.
```

- 14-core Intel Xeon Gold 6132 CPU ($\sim$ 1 200 GFlops):

```
$ srun --gres=gpu:v100:1,gpuexcl ... ./axpy.cpu 500E6
Runtime was 0.184 s.
Floprate was 5 GFlops.
```

- Nvidia Tesla V100 GPU ($\sim$ **7 000** GFlops):

```
$ srun --gres=gpu:v100:1,gpuexcl ... ./axpy.cuda 500E6
Runtime was 0.014 s.
Floprate was 70 GFlops.
```

AXPY example (actual floprate)

The V100 is over 13 times faster than the Xeon but **we are using only 1% of the performance!**

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,

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

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 RAM; or the CUDA cores and the VRAM.

Memory throughput (theoretical memory bandwidth)

- A **theoretical memory bandwidth** can be calculated for each device.

Memory throughput (theoretical memory bandwidth)

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

$$\sim 35 \text{ GB/s}$$

Memory throughput (theoretical memory bandwidth)

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

$$\sim 35 \text{ GB/s}$$

- 14-core Intel Xeon Gold 6132 CPU:

$$\sim 100 \text{ GB/s}$$

Memory throughput (theoretical memory bandwidth)

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

$\sim$ 35 GB/s

- 14-core Intel Xeon Gold 6132 CPU:

$\sim$ 100 GB/s

- Nvidia Tesla V100 GPU:

$\sim$ **900** GB/s

AXPY example (actual memory througput)

- Quad-core Intel Skylake CPU ($\sim$ 35 GB/s):

```
$ ./axpy.cpu 500E6
Runtime was 0.484 s.
Memory throughput 25 GB/s.
```

AXPY example (actual memory througput)

- Quad-core Intel Skylake CPU ($\sim$ 35 GB/s):

```
$ ./axpy.cpu 500E6
Runtime was 0.484 s.
Memory throughput 25 GB/s.
```

- 14-core Intel Xeon Gold 6132 CPU ($\sim$ 100 GB/s):

```
$ srun --gres=gpu:v100:1,gpuexcl ... ./axpy.cpu 500E6
Runtime was 0.184 s.
Memory throughput 65 GB/s.
```

AXPY example (actual memory througput)

- Quad-core Intel Skylake CPU ($\sim$ 35 GB/s):

```
$ ./axpy.cpu 500E6
Runtime was 0.484 s.
Memory throughput 25 GB/s.
```

- 14-core Intel Xeon Gold 6132 CPU ($\sim$ 100 GB/s):

```
$ srun --gres=gpu:v100:1,gpuexcl ... ./axpy.cpu 500E6
Runtime was 0.184 s.
Memory throughput 65 GB/s.
```

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

```
$ srun --gres=gpu:v100:1,gpuexcl ... ./axpy.cuda 500E6
Runtime was 0.014 s.
Memory throughput 845 GB/s.
```

AXPY example (actual memory througput)

We are using between 65% and **95%** of the memory bandwidth!

The AXPY kernel is **memory bound!**

GEMM example (CPU)

- Lets perform a second experiments:

$$\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{n \times n}$$
$$\boldsymbol{C} \leftarrow \boldsymbol{A}\boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{n \times n}$$

GEMM example (CPU)

- Lets perform a second experiments:

$$\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{n \times n}$$
$$\boldsymbol{C} \leftarrow \boldsymbol{AB}, \boldsymbol{C} \in \mathbb{R}^{n \times n}$$

- CPU code would looks like this:

```
double *A = malloc(n*ldA*sizeof(double));
double *B = malloc(n*ldB*sizeof(double));
double *C = malloc(n*ldC*sizeof(double));

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        A[i*ldA+j] = 2.0 * rand()/RAND_MAX - 1.0;

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        B[i*ldB+j] = 2.0 * rand()/RAND_MAX - 1.0;

// compute C <- A * B (level 3 BLAS routine)
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
    n, n, n, 1.0, A, ldA, B, ldB, 0.0, C, ldC);
```

GEMM example (CUDA)

- CUDA code would looks like this:

```
// allocate managed memory
double *A, *B, *C;
cudaMallocManaged(&A, n*ldA*sizeof(double));
cudaMallocManaged(&B, n*ldB*sizeof(double));
cudaMallocManaged(&c, n*ldC*sizeof(double));

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        A[i*ldA+j] = 2.0 * rand()/RAND_MAX - 1.0;

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        B[i*ldB+j] = 2.0 * rand()/RAND_MAX - 1.0;

// prefetch data to GPU memory
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(A, n*ldA*sizeof(double), device, NULL);
cudaMemPrefetchAsync(B, n*ldB*sizeof(double), device, NULL);
cudaDeviceSynchronize();

// initialize cuBLAS
cublasHandle_t handle;
cublasCreate(&handle);

// compute C <- A * B (level 3 BLAS routine)
double alpha = 1.0, beta = 0.0;
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
    n, n, n, &alpha, A, ldA, B, ldB, &beta, C, ldC)
```

GEMM example (actual floprate)

- Quad-core Intel Skylake CPU ($\sim$ 200 GFlops):

```
$ ./gemm.cpu 10000
Runtime was 12.050 s.
Floprate was 166 GFlops.
```

GEMM example (actual floprate)

- Quad-core Intel Skylake CPU ($\sim$ 200 GFlops):

```
$ ./gemm.cpu 10000
Runtime was 12.050 s.
Floprate was 166 GFlops.
```

- 14-core Intel Xeon Gold 6132 CPU ($\sim$ 1 200 GFlops):

```
$ srun --gres=gpu:v100:1,gpuexcl ...  ./gemm.cpu 10000
Runtime was 2.250 s.
Floprate was 889 GFlops.
```

GEMM example (actual floprate)

- Quad-core Intel Skylake CPU ($\sim$ 200 GFlops):

```
$ ./gemm.cpu 10000
Runtime was 12.050 s.
Floprate was 166 GFlops.
```

- 14-core Intel Xeon Gold 6132 CPU ($\sim$ 1 200 GFlops):

```
$ srun --gres=gpu:v100:1,gpuexcl ...  ./gemm.cpu 10000
Runtime was 2.250 s.
Floprate was 889 GFlops.
```

- Nvidia Tesla V100 GPU ($\sim$ **7 000** GFlops):

```
$ srun --gres=gpu:v100:1,gpuexcl ... ./gemm.cuda 10000
Runtime was 0.308 s.
Floprate was 6503 GFlops.
```

GEMM example (actual floprate)

We are using between 74% and **92%** of the floating-point performance!

The GEMM kernel is **compute bound!**

Arithmetical intensity (definition)

- How do we know which kernels are memory bound and which are compute bound?

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 AXPY has the arithmetical intensity of

$$\text{Arithmetical intensity}_{\text{AXPY,double}} = \frac{2 \text{ Flop}}{3 \cdot 8 \text{ Byte}} = \frac{1}{12} \text{ Flop/Byte}.$$

Arithmetical intensity (examples)

- Double precision AXPY has the arithmetical intensity of

$$\text{Arithmetical intensity}_{\text{AXPY,double}} = \frac{2 \text{ Flop}}{3 \cdot 8 \text{ Byte}} = \frac{1}{12} \text{ Flop/Byte.}$$

- Single precision AXPY has the arithmetical intensity of

$$\text{Arithmetical intensity}_{\text{AXPY,single}} = \frac{2 \text{ Flop}}{3 \cdot 4 \text{ Byte}} = \frac{1}{6} \text{ Flop/Byte.}$$

Arithmetical intensity (examples)

- Double precision AXPY has the arithmetical intensity of

$$\text{Arithmetical intensity}_{\text{AXPY,double}} = \frac{2 \text{ Flop}}{3 \cdot 8 \text{ Byte}} = \frac{1}{12} \text{ Flop/Byte.}$$

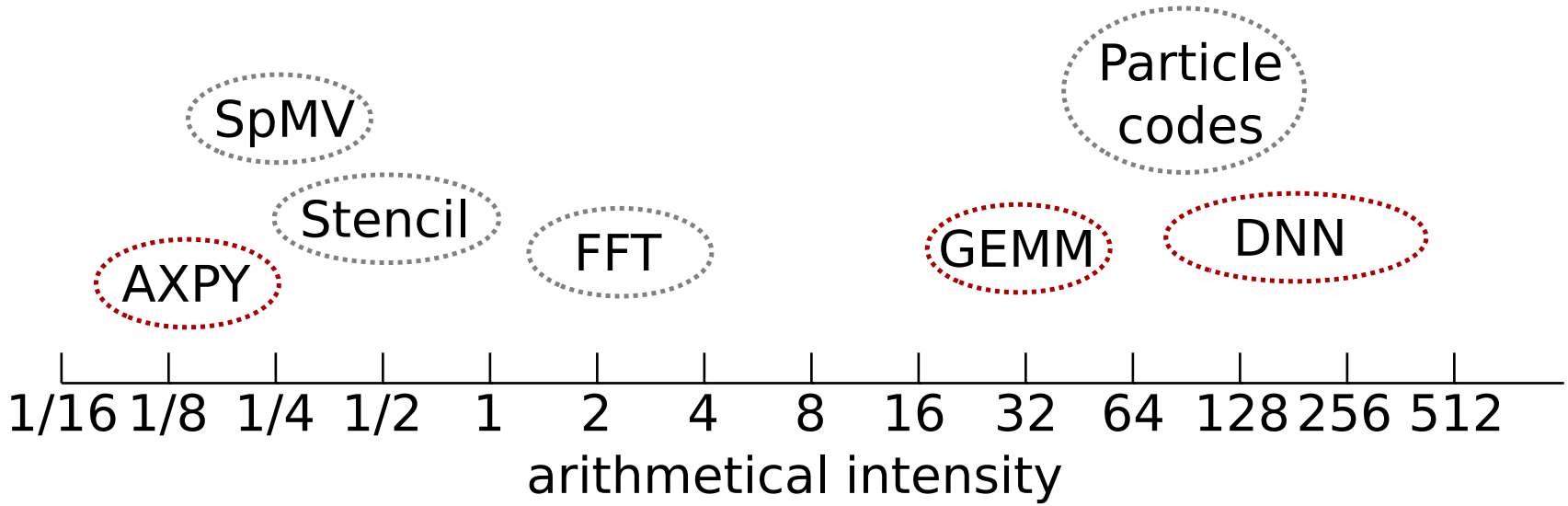
- Single precision AXPY has the arithmetical intensity of

$$\text{Arithmetical intensity}_{\text{AXPY,single}} = \frac{2 \text{ Flop}}{3 \cdot 4 \text{ Byte}} = \frac{1}{6} \text{ Flop/Byte.}$$

- Double precision GEMM has the arithmetical intensity of

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

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 (Deep Neural Networks)

- Estimated **single precision** numbers:

Operation	**Arithmetical intensity**
Linear layer (4096 outputs, 1024 inputs, **batch size 512**)	**158** Flop/Byte
Linear layer (4096 outputs, 1024 inputs, batch size 1)	0.5 Flop/Byte
Max pooling with 3x3 window and unit stride	1.125 Flop/Byte
ReLU activation	0.125 Flop/Byte
Layer normalization	< 5 Flop/Byte

Arithmetical intensity (Deep Neural Networks)

- Estimated **double precision** numbers:

Operation	**Arithmetical intensity**
Linear layer (4096 outputs, 1024 inputs, **batch size 512**)	**79** Flop/Byte
Linear layer (4096 outputs, 1024 inputs, batch size 1)	0.25 Flop/Byte
Max pooling with 3x3 window and unit stride	0.56 Flop/Byte
ReLU activation	0.06 Flop/Byte
Layer normalization	$<$ 2.5 Flop/Byte

Arithmetical intensity (optimal intensity)

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

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

Arithmetical intensity (optimal intensity)

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

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

- If the arithmetical intensity is **smaller than** the optimal intensity, the kernel is **memory bound**.

Arithmetical intensity (optimal intensity)

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

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

- If the arithmetical intensity is **smaller than** the optimal intensity, the kernel is **memory bound**.
- If the arithmetical intensity is **larger than** the optimal intensity, the kernel is **compute bound**.

Arithmetical intensity (optimal intensity, double precision)

- Quad-core Intel Skylake CPU:

$$\sim 5.7 \text{ Flop/Byte}$$

Arithmetical intensity (optimal intensity, double precision)

- Quad-core Intel Skylake CPU:

$\sim$ 5.7 Flop/Byte

- 14-core Intel Xeon Gold 6132 CPU:

$\sim$ 12 Flop/Byte

Arithmetical intensity (optimal intensity, double precision)

- Quad-core Intel Skylake CPU:

$\sim$ 5.7 Flop/Byte

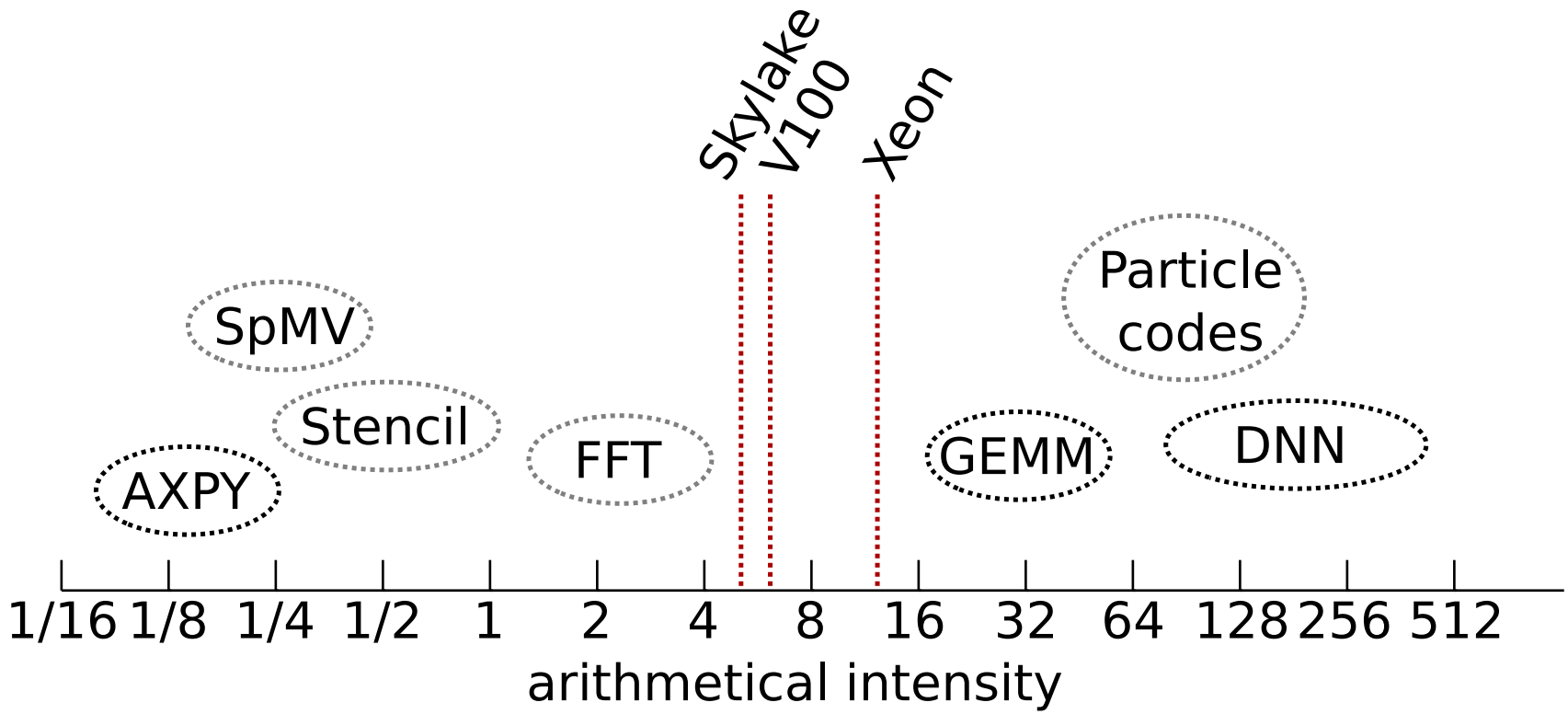
- 14-core Intel Xeon Gold 6132 CPU:

$\sim$ 12 Flop/Byte

- Nvidia Tesla V100 GPU:

$\sim$ 7.7 Flop/Byte

Arithmetical intensity (optimal intensity, double precision)



Arithmetical intensity (optimal intensity, single precision)

- Quad-core Intel Skylake CPU:

$\sim$ 11.4 Flop/Byte

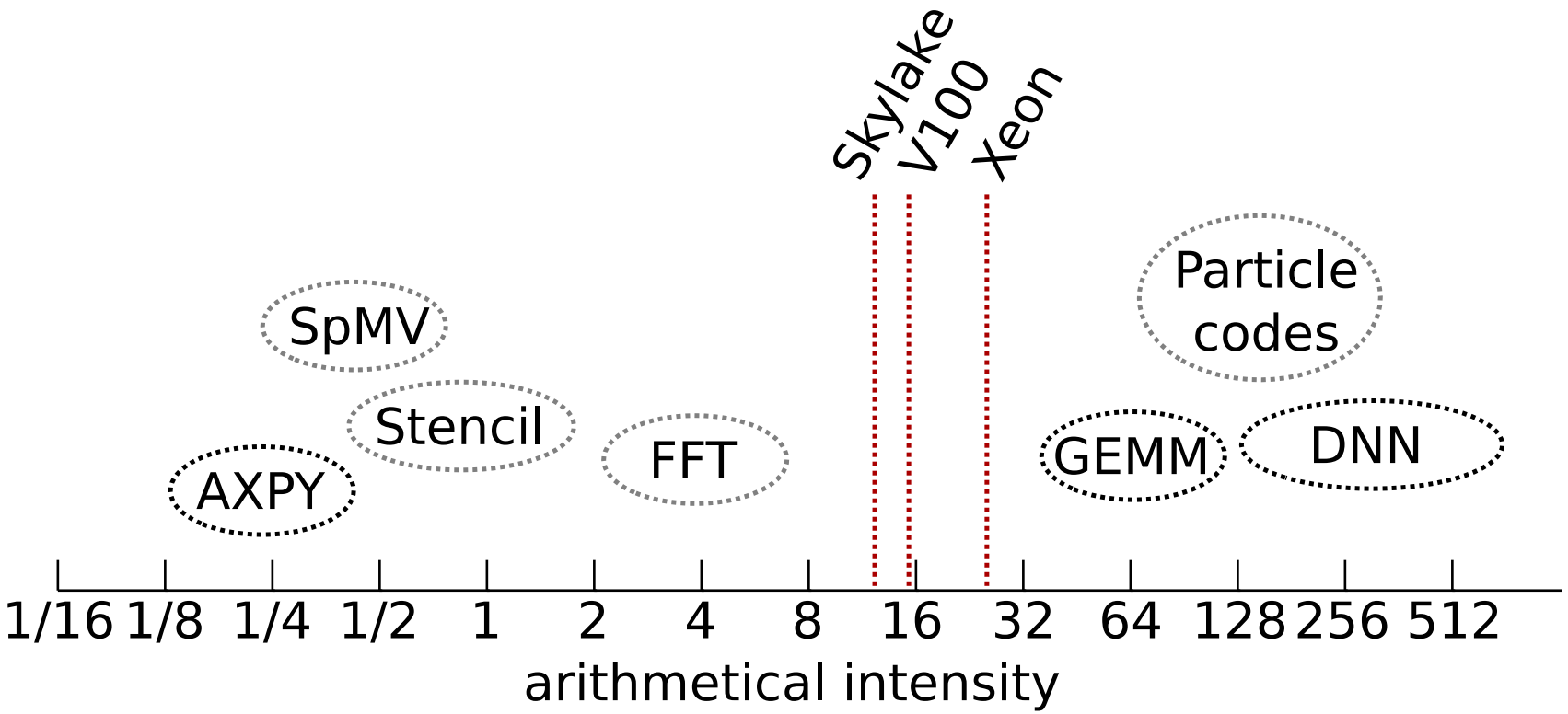
- 14-core Intel Xeon Gold 6132 CPU:

$\sim$ 24 Flop/Byte

- Nvidia Tesla V100 GPU:

$\sim$ 15.6 Flop/Byte

Arithmetical intensity (optimal intensity, double precision)



Arithmetical intensity (optimal intensity, single precision)

- Quad-core Intel Skylake CPU:

  — Flop/Byte

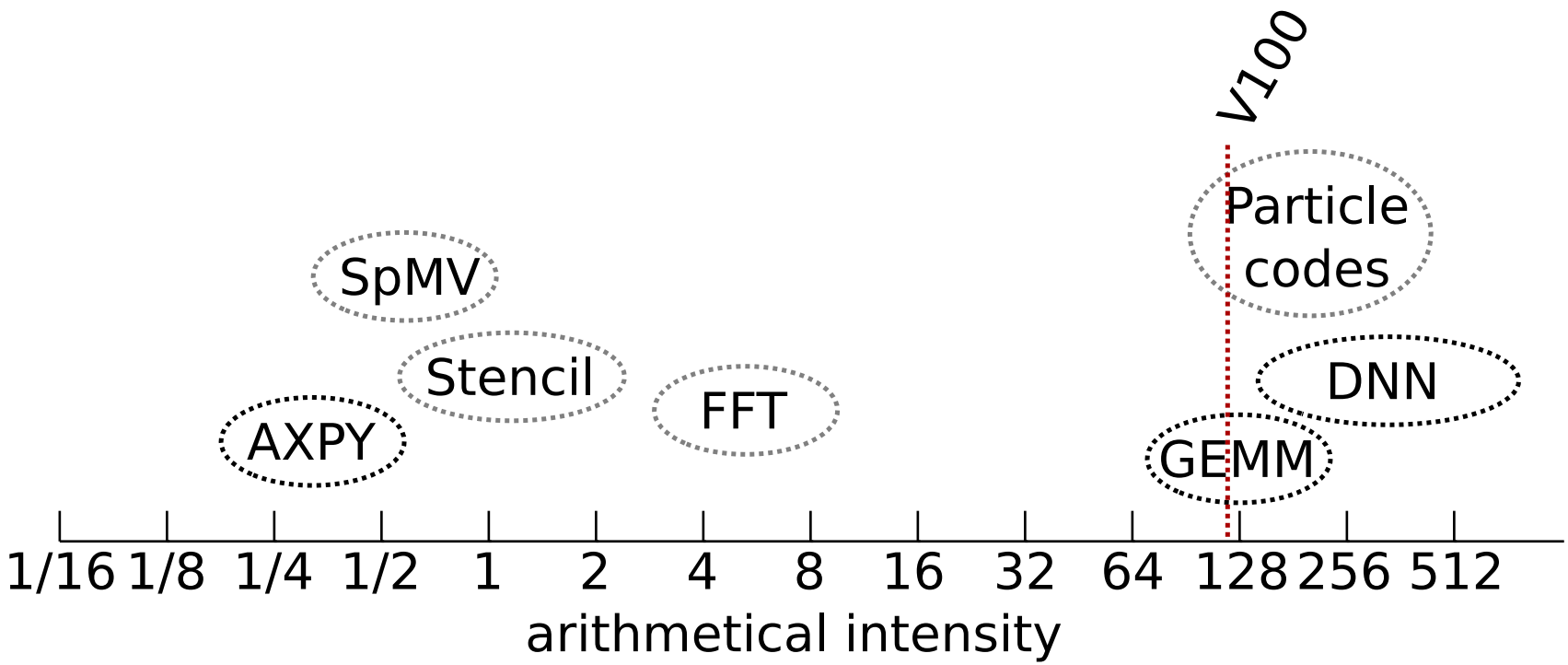
- 14-core Intel Xeon Gold 6132 CPU:

  — Flop/Byte

- Nvidia Tesla V100 GPU:

  $\sim$ **124** Flop/Byte

Arithmetical intensity (optimal intensity, half precision)



Arithmetical intensity (Caches and shared memory)

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

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

Arithmetical intensity (Caches and shared memory)

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

$$\text{Arithmetical intensity}_{\text{GEMM,double}} = \frac{2n-1}{8(2n+1)} \text{ Flop/Byte}$$
$$= \sim \frac{1}{8} \text{ Flop/Byte}$$

- Why is it

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

Arithmetical intensity (Caches and shared memory)

- When implemented naively, we compute each entry separately:

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

Arithmetical intensity (Caches and shared memory)

- When implemented naively, we compute each entry separately:

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

- However, we can also do the following:

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

Arithmetical intensity (Caches and shared memory)

- If m is small enough, $\boldsymbol{A}_{ik}$ and $\boldsymbol{B}_{kj}$ can be fitted into CPU caches or SMP's shared memory.

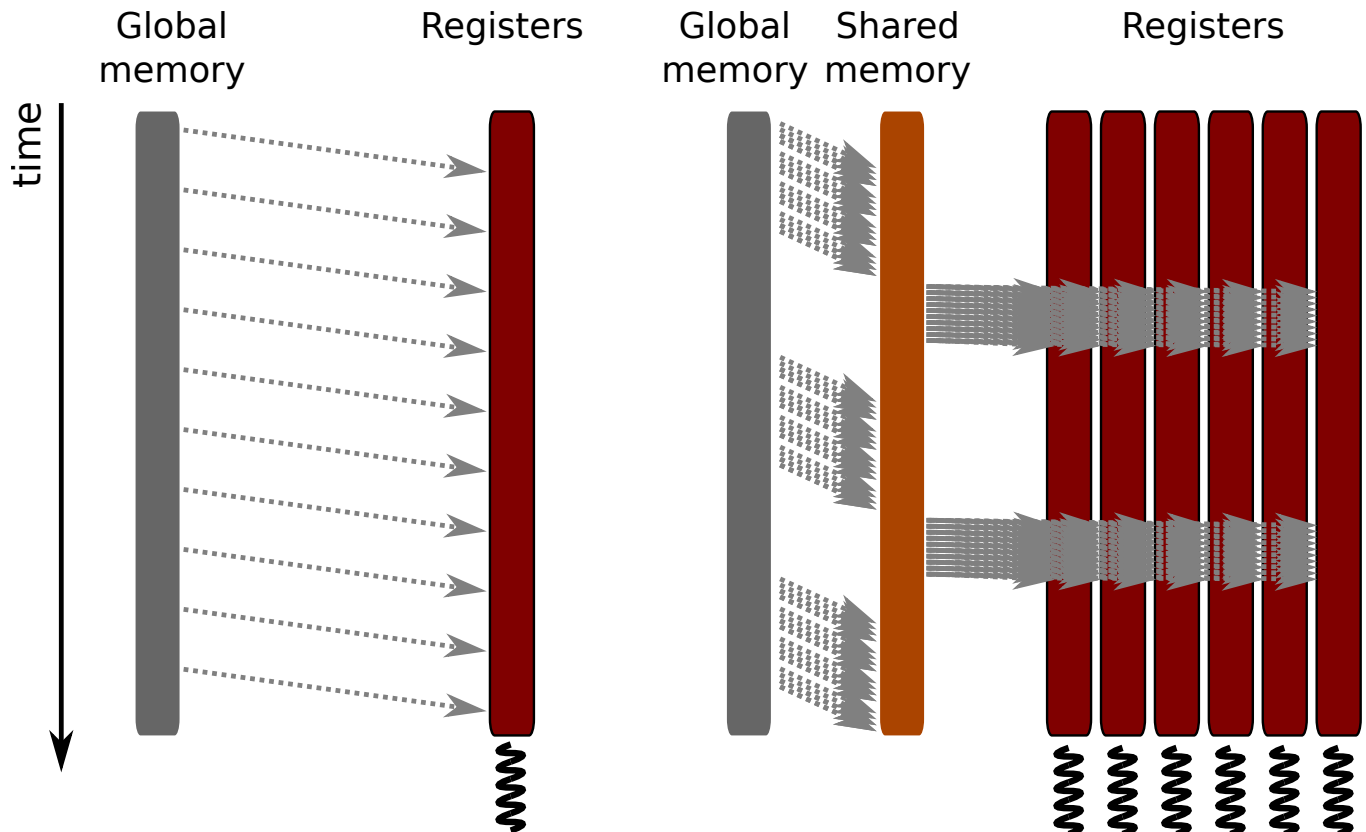
Arithmetical intensity (Caches and shared memory)

- If m is small enough, $\boldsymbol{A}_{ik}$ and $\boldsymbol{B}_{kj}$ can be fitted into CPU caches or SMP's shared memory.
- Each block is **shared among the thread block**!

Arithmetical intensity (Caches and shared memory)

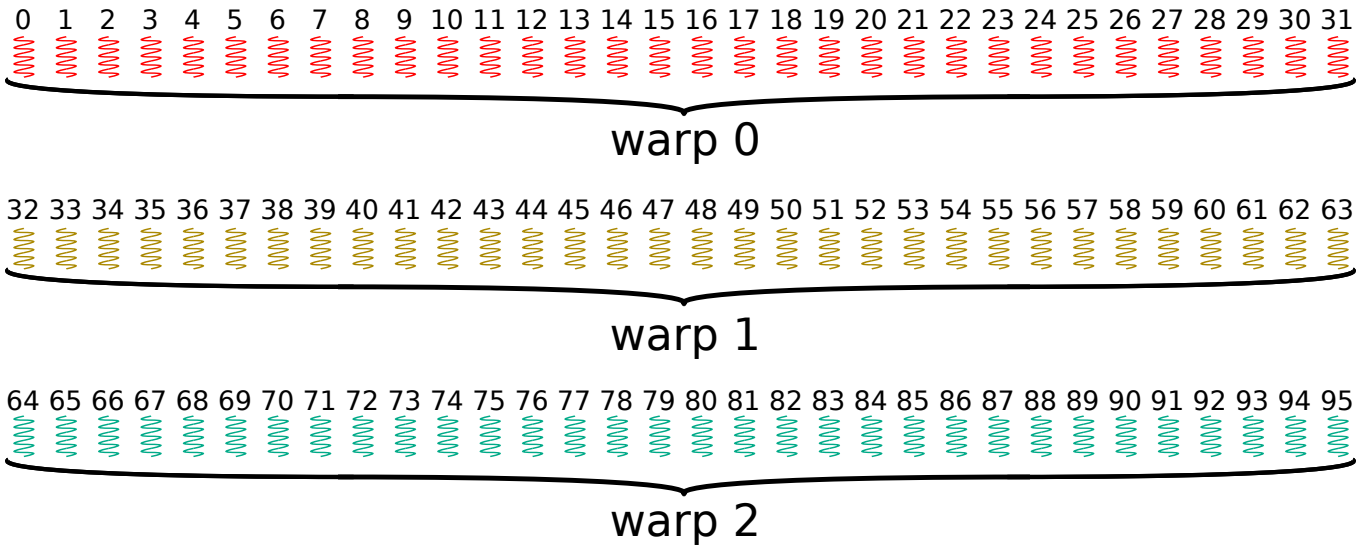
- If m is small enough, $\boldsymbol{A}_{ik}$ and $\boldsymbol{B}_{kj}$ can be fitted into CPU caches or SMP's shared memory.
- Each block is **shared among the thread block**!

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



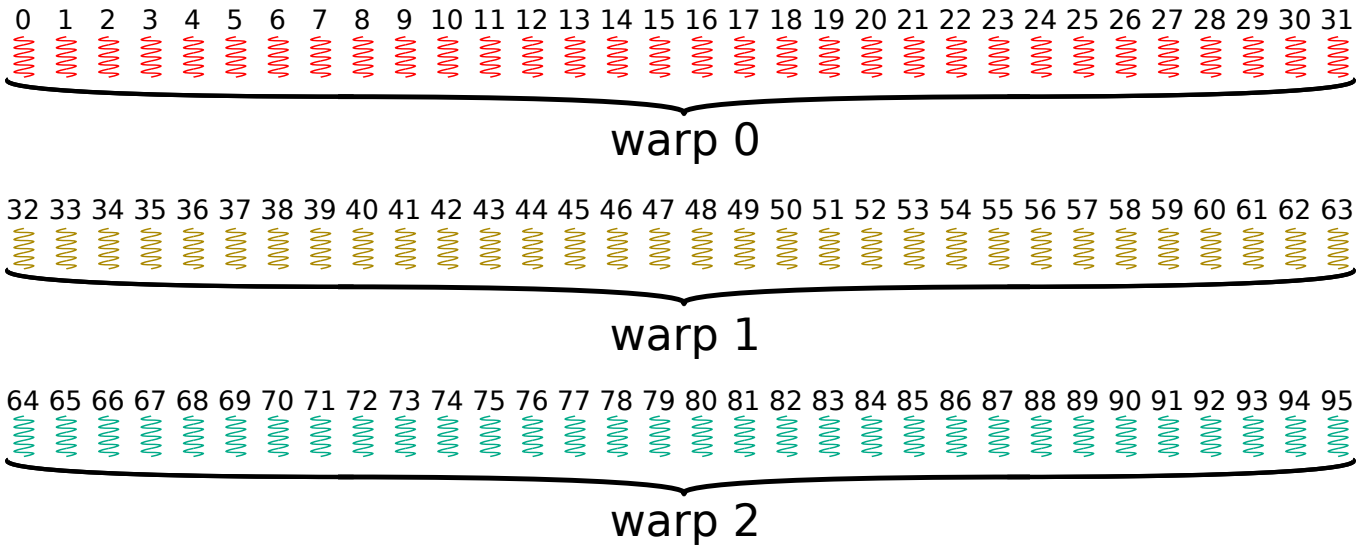
Warps

- The GPU hardware divisions each thread block into sub-groups called **warps**:



Warps

- The GPU hardware divisions each thread block into sub-groups called **warps**:



- Each warp consists of 32 threads and all of them **are scheduled together**.

Warps (diverging paths)

- The fact the all threads within a warp are scheduled together causes problems:

```
if (threadIdx.x % 2 == 0) {
    // all threads within the warp enter, only even numbered threads
        commit the result
}
else {
    // all threads within the warp enter, only odd numbered threads commit
        the result
}
```

Warps (diverging paths)

- The fact the all threads within a warp are scheduled together causes problems:

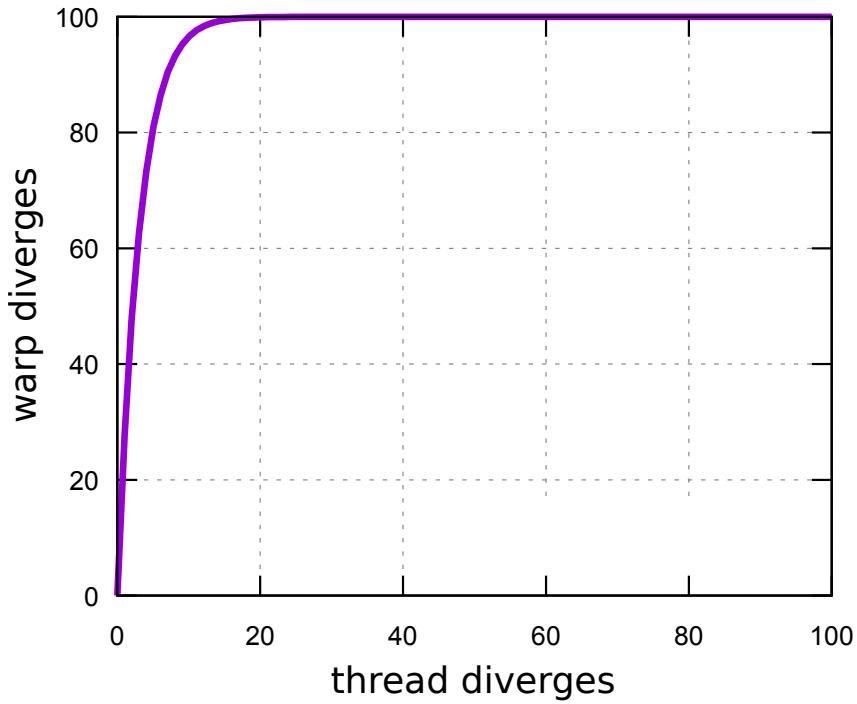
```
if (threadIdx.x % 2 == 0) {
    // all threads within the warp enter, only even numbered threads
          commit the result
}
else {
    // all threads within the warp enter, only odd numbered threads commit
           the result
}
```

- **The cost is the same as if all threads executed both branches!**

Warps (cost of diverging paths)

- If a single thread diverges with the probability $p \in [0, 1]$, then probability that at least one thread within a warp diverges is

$$1 - (1 - p)^{32}$$



PCI-E bandwidth (AXPY example)

- Remember this:

```
// allocate managed memory
double *x, *y;
cudaMallocManaged(&x, n*sizeof(double));
cudaMallocManaged(&y, n*sizeof(double));

for (int i = 0; i < n; i++) {
    x[i] = 2.0 * rand()/RAND_MAX - 1.0;
    y[i] = 2.0 * rand()/RAND_MAX - 1.0;
}

// prefetch data to GPU memory
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(x, n*sizeof(double), device, NULL);
cudaMemPrefetchAsync(y, n*sizeof(double), device, NULL);
cudaDeviceSynchronize();

// initialize cuBLAS
cublasHandle_t handle;
cublasCreate(&handle);

// compute y <- 2 * x + y (level 1 BLAS routine)
double alpha = 2.0;
cublasDaxpy(handle, n, &alpha, x, 1, y, 1);
```

PCI-E bandwidth (AXPY performance)

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

```
$ srun ... ./axpy.cuda 500E6
Runtime was 0.014 s.
Floprate was 70 GFlops.
Memory throughput 844 GB/s.
```

PCI-E bandwidth (comment out prefetch lines)

- Lets comment out some lines:

```
// allocate managed memory
double *x, *y;
cudaMallocManaged(&x, n*sizeof(double));
cudaMallocManaged(&y, n*sizeof(double));

for (int i = 0; i < n; i++) {
    x[i] = 2.0 * rand()/RAND_MAX - 1.0;
    y[i] = 2.0 * rand()/RAND_MAX - 1.0;
}

// prefetch data to GPU memory
// int device = -1;
// cudaGetDevice(&device);
// cudaMemPrefetchAsync(x, n*sizeof(double), device, NULL);
// cudaMemPrefetchAsync(y, n*sizeof(double), device, NULL);
// cudaDeviceSynchronize();

// initialize cuBLAS
cublasHandle_t handle;
cublasCreate(&handle);

// compute y <- 2 * x + y (level 1 BLAS routine)
double alpha = 2.0;
cublasDaxpy(handle, n, &alpha, x, 1, y, 1);
```

PCI-E bandwidth (AXPY performance without prefetch)

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

```
$ srun ... ./axpy.cuda 500E6
Runtime was 1.462 s.
Floprate was 1 GFlops.
Memory throughput 8 GB/s.
```

PCI-E bandwidth (bandwidth)