HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 21

Fabian Wermelinger
Harvard University
CS205

Tuesday, April 25th 2023

LAST TIME

- Explicit versus implicit vectorization (not the same as auto-vectorization)
- The Intel Implicit SPMD Program Compiler (ISPC)
- ISPC execution model (reading assignment)
- How is "SPMD-to-SIMD" mapped to hardware (execution mask in ISPC)
- Basic C/C++ language extensions in ISPC

TODAY

Main topic: ***GPU architectures, Little's Law, large throughput***

Details:

- Differences between CPU and GPU architectures
- Revisit discussion about latency and throughput (lecture 2)
- Latency and throughput in a shopping mall
- Little's Law
- SIMD, SPMD and SIMT
- Nvidia Volta GPU architecture
- Introduction to CUDA
- What we have not discussed in this introduction

WHY GPUS?

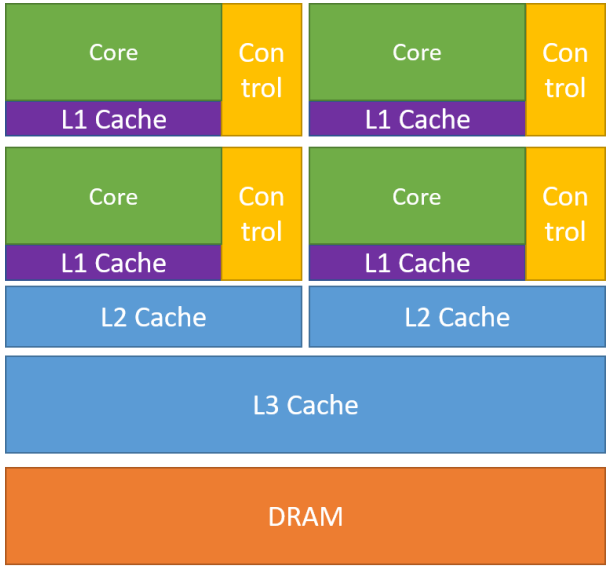
- ***Recall lecture 1:*** Moore's Law is about to end (difficult to increase transistor density and clock rate without material meltdown).
- Since about 2004 → multicore era (increase number of transistors by increasing core count).
- GPUs always were data parallel designs → rendering pixels is massively data parallel.
- GPGPU (*general-purpose GPU*) to use this processing power for other applications than graphics → Nvidia introduced CUDA in 2006 to program their GPUs for general purpose use.

CPU core:	***GPU core:***
• Heavyweight	• Lightweight
• *Cache oriented*	• Runs at *lower* clock rates
• More transistors dedicated to control and logic (good all-rounder core)	• More transistors dedicated to data processing (less flexible core)
• Out-of-order execution	• Requires *much higher memory bandwidth* to service all the cores!
• Speculative execution	

WHY GPUS?

A GPU provides *much higher instruction throughput and memory bandwidth* than a CPU *within a similar price and power envelope.*

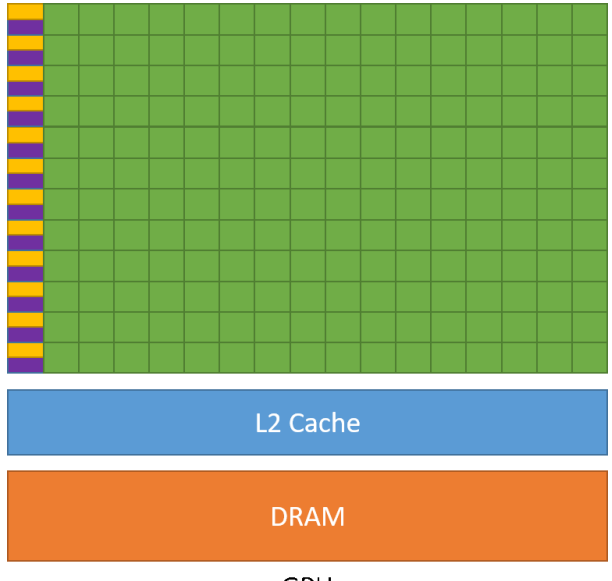
CPU architecture:



CPU

- Optimized for low-latency tasks with heavy usage of fast cache hierarchies.
- More transistors dedicated for control and logic.
- Fat cores.

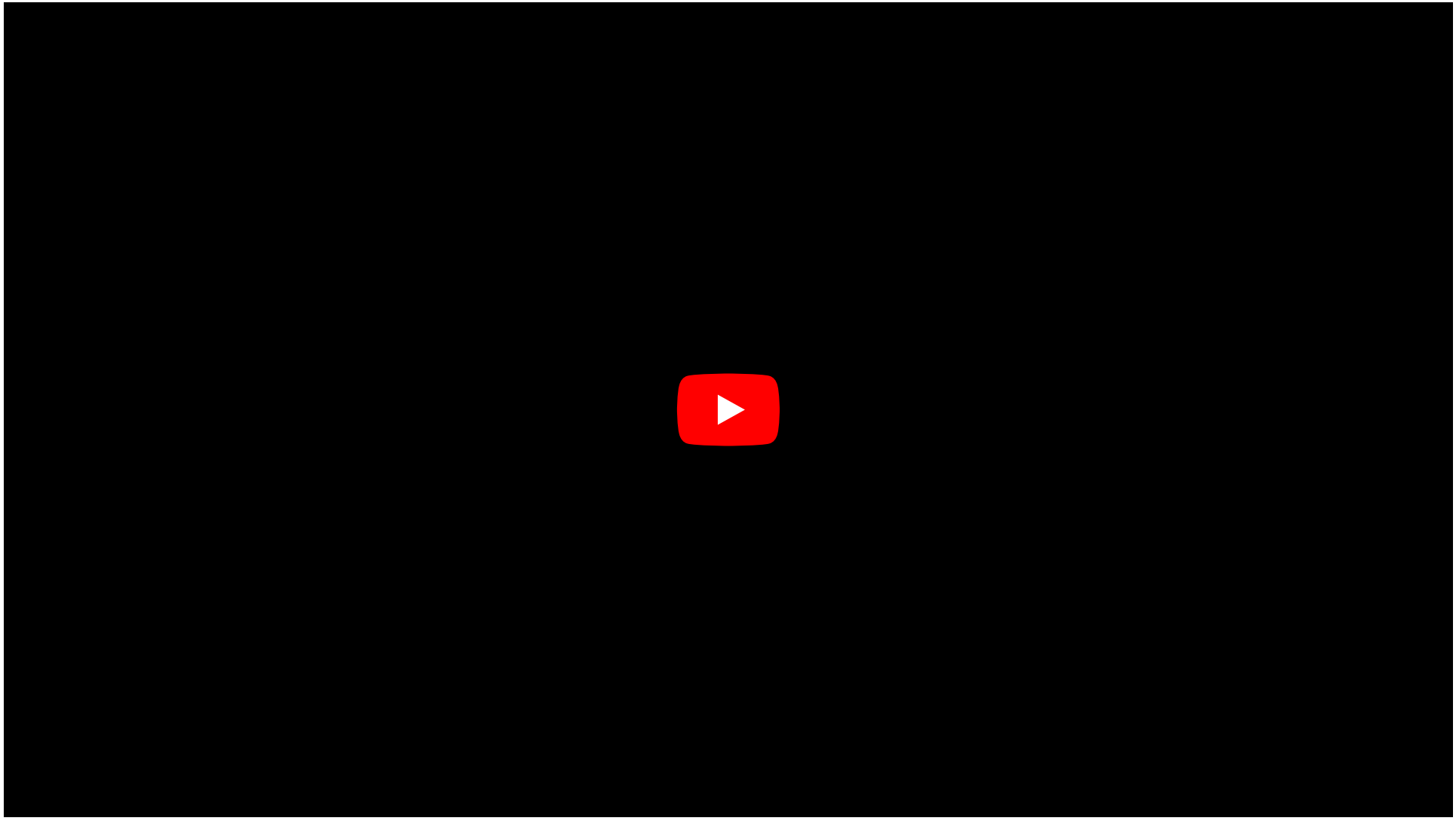
GPU architecture:



GPU

- Optimized for high instruction throughput.
- More transistors dedicated to data processing (many light cores).
- Associated with high latencies → ***how are they hidden on GPUs? Will this work for any application?***

WHY GPUS?



https://www.youtube.com/watch?v=-P28LKWTzrI

LATENCY AND THROUGHPUT

Recall discussion from lecture 2:

- ***Latency:*** the *response time* for information to arrive measured from its initial request. Latency is often related to physical parameters → the time to fill the pipe is proportional to its *dimension*.
- ***Bandwidth:*** the *throughput* of a subsystem. Naively the inverse of latency *(neglecting concurrency)*. Modern subsystems can *overlap multiple requests* to increase bandwidth! → bandwidth is more generally defined as the *rate at which requests can be satisfied*.

Latency:

- Latency *denotes a waiting time*. Latency is given in units of time.
- Does not depend on the workload.
- Characteristics of *low latency*: low mass (inertia); agile; adaptivity; offense in sports.
- Characteristics of *high latency*: high mass (inertia); sluggish; defense in sports.
- CPU architectures are optimized for latency.

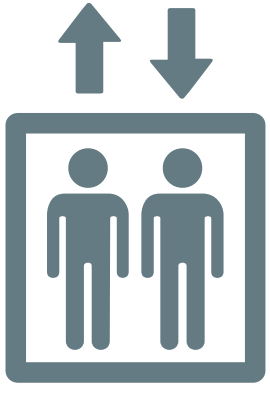
Throughput:

- Throughput *denotes a rate-based derived metric* (lecture 15).
- For the pipe example above, *bandwidth* and *throughput* are interchangeable.
- The throughput tells you how long it takes to process a certain workload.
- GPU architectures are optimized for high throughput.

LATENCY AND THROUGHPUT

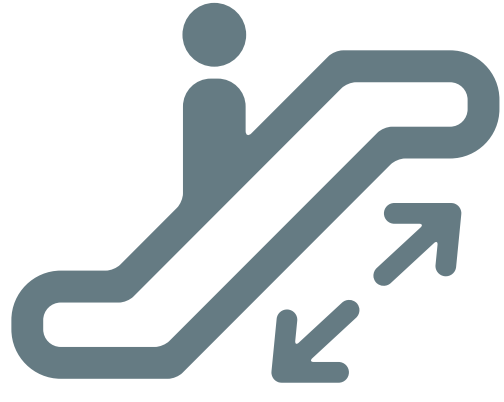
Latency:

- Latency *denotes a waiting time*. Latency is given in units of time.
- Does not depend on the workload.
- Characteristics of *low latency*: low mass (inertia); agile; adaptivity; offense in sports.
- Characteristics of *high latency*: high mass (inertia); sluggish; defense in sports.
- CPU architectures are optimized for latency.



Throughput:

- Throughput *denotes a rate-based derived metric* (lecture 15).
- For the pipe example above, *bandwidth* and *throughput* are interchangeable.
- The throughput tells you how long it takes to process a certain workload.
- GPU architectures are optimized for high throughput.



LATENCY AND THROUGHPUT (MORE EXAMPLES)

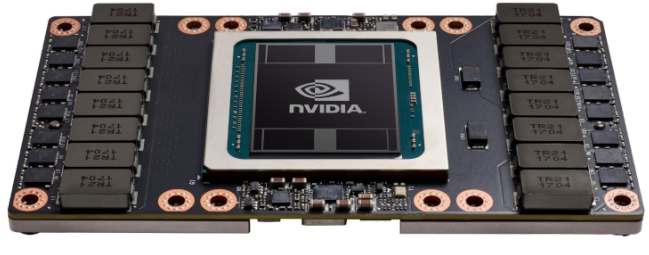
Industrial presses for wooden plates:

Workshop press:

Continuous press (plant):

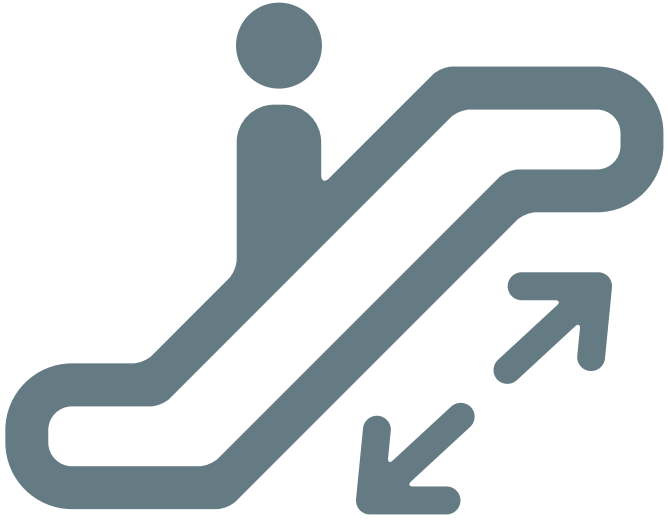
Optimized for latency → can work on many different plate formats and can put them in and out of the press quickly.

Optimized for throughput → takes time to change plate format. Once setup you get *a lot* of plates per unit time.



IS ALL WAITING TIME LATENCY?

Assume an escalator can output 2 person/s (rate is "person per second")



- From the given throughput, we can infer a *per-person timescale* of 0.5 seconds.
- This timescale *is not a latency* but a *characteristic time* for the escalator to output one person.
- ***Is this timescale a meaningful quantity for escalators in general?*** Only if the escalator has a *single lane*.

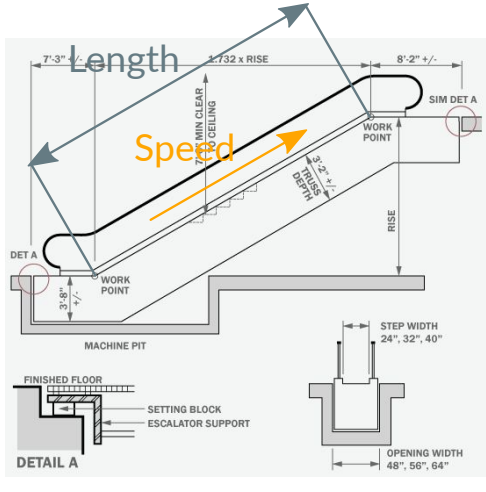
CHARACTERISTIC TIME OF ESCALATORS

Characteristic time of escalators:

- Assume throughput is again 2 person/s
- Escalator has 2 lanes now
- The characteristic timescale of that system is *1 second* (time to process two work units).
- These timescales are not latencies.

Escalator latency:

- Do escalators have latency?
- ***Recall:*** latency is *independent* of the workload.
- The escalator latency is the time it takes to fill its *characteristic length* with work units



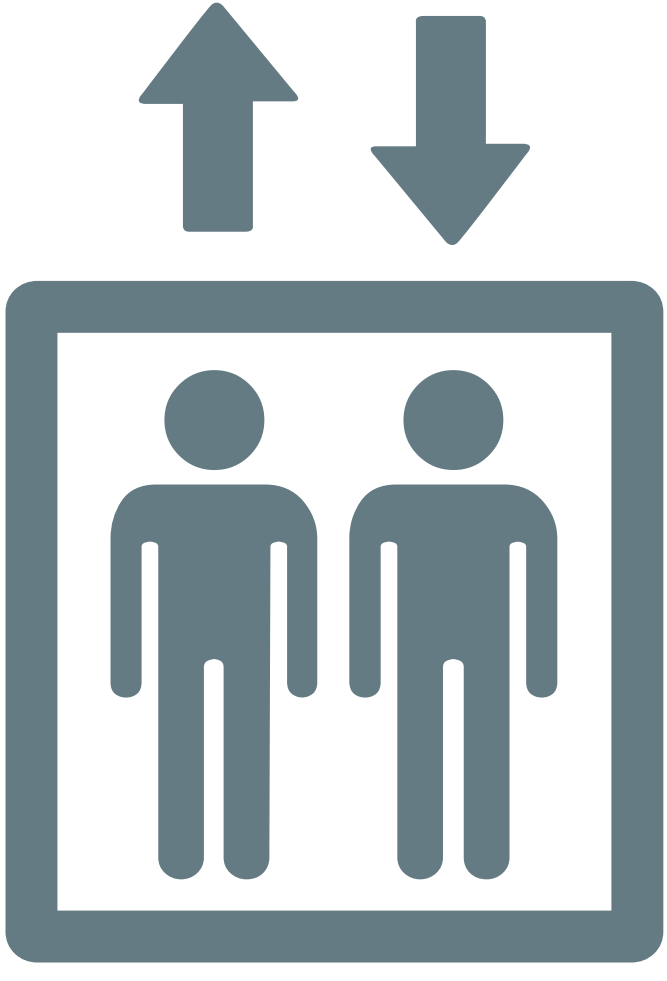
Escalator latency:

$$\frac{\text{Length [m]}}{\text{Speed [m/s]}}$$

High bandwidth devices may have a very high latency.

ELEVATOR THROUGHPUT

Do elevators have throughput?



- Assume a one-way elevator trip latency of 8 seconds.
- Assume the elevator has a capacity of 8 persons.
- The *unidirectional* throughput is 8 persons / 16 seconds = 0.5 person/s
- For *latency dominated* devices the throughput is *capacity divided by latency*.
- ***How meaningful is the quantity "throughput" for an elevator?***

LATENCY OR THROUGHPUT?

- *Software targeting high-throughput:*
  - Tries to optimize the execution of a task considering that the *workload diverges to infinity*.
  - Very small workloads are just ignored.
  - HPC and supercomputing pursues mostly high-throughput software.
- *Software targeting low-latency:*
  - Tries to optimize the execution of a task considering realistic/minimal workloads and bad scenarios.
  - Small workloads are considered important.
  - Huge workloads are not considered.
  - Examples for low-latency software are real-time systems or games.

Both *high-throughput* and *low-latency* strategies have the same goal: ***decrease time-to-solution***. The difference between them is the regime in which they operate.

EXAMPLE: EMPTY SHOPPING MALL

- Consider 3 friends in an empty shopping mall and you want to go one level up.
- Elevator option:
  - Latency: 8 seconds
  - Capacity: 8 persons
- Escalator option:
  - 2 lanes
  - Latency: 15 seconds
  - Throughput: 2 persons/second

Which is the better option in this scenario: elevator or escalator?

- ***Elevator:*** 8 seconds (***best option***)
- ***Escalator:*** 15 + 1 = 16 seconds

EXAMPLE: BUSY SHOPPING MALL

- Consider 3000 friends in a shopping mall. Everybody want to go one level up and there is no external contention:

  Elevator time-to-solution:

  $$8\,\mathrm{s} + \frac{3000 - 8}{8\,\mathrm{s}} \times 16\,\mathrm{s} = 5992\,\mathrm{s}$$

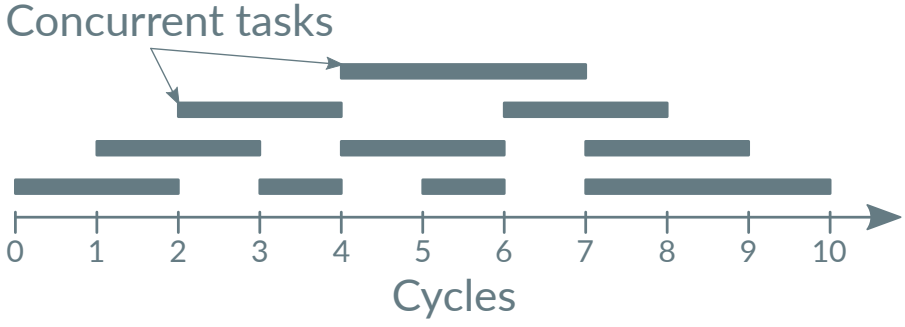
  Escalator time-to-solution:

  $$15\,\mathrm{s} + \frac{3000 - 2}{2\,\mathrm{persons/s}} = 1514\,\mathrm{s}$$

- Escalator is 4 times faster! ***Why?***
- ***Nominal throughput:***
  - ***Elevator:*** 0.5 person/second
  - ***Escalator:*** 2 person/second

LITTLE'S LAW

Concurrent process:



3 metrics:

1. ***Latency:*** here from 1 to 3 cycles with *mean latency* 2 cycles.
2. ***Throughput:*** number of tasks processed in the interval. Here 10 tasks in 10 cycles → throughput 1 task/cycle.
3. ***Concurrency:*** number of tasks processed at the same time. Ranges from 1 to 3 with *mean concurrency* of 2 tasks.

Little's Law:

mean concurrency = mean latency × throughput

Occupancy (utilization):

Occupancy = instantaneous concurrency / mean concurrency

→ ***Shopping mall:*** need at least *30 people* for 100% utilization of one 2-lane escalator!

LITTLE'S LAW EXAMPLE

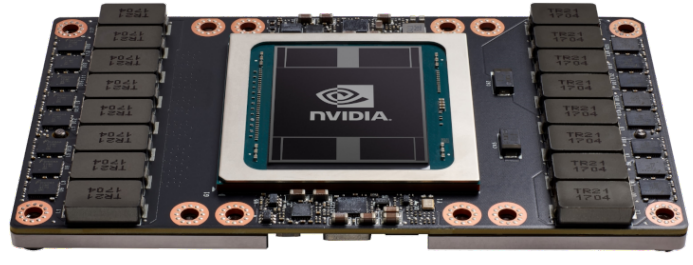
Consider two architectures:

Intel Xeon E5-2683v4 CPU:

- Base frequency of 2100 MHz and 76.8 GB/s bandwidth
- About *200 cycles latency* to access memory (recall lecture 2). This translates to *95 ns* latency for memory access.

Nvidia Tesla V100 GPU:

- Base frequency of 1370 MHz and 900 GB/s bandwidth
- About *400 cycles latency* to access memory (high bandwidth devices usually have higher latency). This translates to *292 ns* latency for memory access.



Mean concurrency: *(for a memory bound application)*

$7296 \text{ byte} = 76.8 \times 10^9 \text{ byte/s} \times 95 \times 10^{-9} \text{ s}$

$262800 \text{ byte} = 900 \times 10^9 \text{ byte/s} \times 292 \times 10^{-9} \text{ s}$

You need at least this many bytes in-flight at any given time to maintain 100% utilization (occupancy) of the device.

The ***GPU requires 36×*** more parallelism than the CPU for full occupancy (12× more due to higher throughput and 3× more to *hide* the longer latency)! Not all applications can offer this additional parallelism and will therefore not run efficiently on the GPU.

FROM THE SHOPPING MALL TO THE CPU

How do these concepts relate to what we learned for CPU architectures?

Shopping mall	CPU
People	Instructions
Seconds	Cycles
Single-lane escalator	5-stage pipelined CPU core (IF–ID–EX–MEM–WB)
Multi-lane escalator	Contemporary CPU core (e.g. Xeon E5-2683v4)
Escalator lanes of the same type	SIMD lanes
Elevator	Non-pipelined CPU or microcontroller
Differentiation run/walk/stand on lanes	Out-of-order execution ports (*superscalar*)

- *Little's Law* applies within CPU or GPU components (and in shopping malls, pressing plants for wood products and beyond).
- *Occupancy* is connected to Little's Law and provides valuable insight for resource utilization.

SIMD, SPMD AND SIMT

Recall ISPC: the *key difference* between SIMD and the SPMD model discussed with ISPC is that *SIMD exposes the vector width to the software*.

- ***Writing explicit SIMD code with Intel intrinsics:*** you are working with types that resemble vector registers (e.g. `__m128`) and *you must think about all SIMD lanes simultaneously*.
- ***Writing implicit SIMD code with ISPC:*** you put yourself in the position of *one SIMD lane*. We called each SIMD lane a *program instance* (SPMD) and a group of program instances a gang.
- The SPMD model is also used for (Nvidia) GPU programming. Here the program instance is a *thread*, each with a program counter and a call stack. To differentiate this execution model from both SIMD (not what exactly happens on GPU) and SPMD (too general), the term ***S**ingle–**I**nstruction–**M**ultiple–**T**hread* (SIMT) is used. ***Note:*** *SIMT hardware combines DLP + TLP at runtime. In ISPC (or SIMD) TLP is complementary.* ***The GPU is a shared memory architecture***.

- ***CUDA*: *C*ompute *U*nified *D*evice *A*rchitecture** is a programming model for GPGPU programming with support for C/C++/Fortran programming languages. CUDA ships with a large API for host/device coordination.
- *Host* denotes the architecture that hosts the GPU, commonly a CPU.
- *Device* denotes the GPU.
- A *kernel* is a CUDA program (function) that is executed *on each CUDA thread* (***recall:*** ISPC functions are executed by each program instance).
- *CUDA threads* are software threads that each execute the same kernel function. Since compute capability 7.x CUDA threads maintain individual program counters.
- A *warp* is a collection of 32 CUDA threads executed together (similar to SIMD width). Warps are scheduled for execution on a *streaming multiprocessor* similar to a physical thread. Before compute capability 7.x each warp maintained a program counter, similar to a *gang* in ISPC.

STREAMING MULTIPROCESSORS

- The streaming multiprocessor (SM) is the main processing unit on Nvidia GPUs.
- The design of the SM changes frequently among GPU generations. The image on the right corresponds to the Nvidia Volta architecture (GV100).
- The SM has *4 processing blocks*, each with 16 32-bit float, 8 64-bit float, 16 32-bit integer cores and 2 mixed-precision tensor cores (for DL matrix arithmetic), one *warp scheduler* and a 64 KB register file.
- Volta GPUs have *80 SM's* in total.
- Volta GPUs are clocked at 1530 MHz and support FMA instructions. ***Nominal peak performance for 32-bit floats:***
  cores × clock × flop/cycle
  80×4×16 × 1.530 × 2 = ***15667 Gflop/s***

STREAMING MULTIPROCESSORS

- Each of the 4 processing blocks can *issue one instruction per warp in every cycle*. For the 80 SM's on the Volta GPU, that is a *throughput* of 320 instructions/cycle.
- On Volta, arithmetic instructions (e.g. FMA) have a *4 cycle latency*.
- Little's Law tells us that you need at least *1280 independent* instructions to hide this latency. Independent means no dependencies on instruction input operands (registers) and all data available in registers (memory access is $\mathcal{O}(100)$ cycles latency).
- This concurrency can be provided by 1280 *independent warps (TLP)* or fewer warps where each warp has *multiple independent instructions to execute (ILP)* (e.g. unrolling in kernel).
- ***Caches are the key concept on the CPU that help us hide these latencies. On the GPU caching is explicit and instead we must provide more parallelism to hide these latencies!***

NVIDIA CUDA: VECTOR ADDITION

Consider the operation: $y = y + x$ with $x, y \in \mathbb{R}^n$

C++ kernel:

```cpp
void add(int n, float *x, float *y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
```

add.cpp

CUDA kernel (naive):

```cpp
__global__ void add(int n, float *x, float *y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
```

add.cu

Function execution space specifiers:

- `__global__`: declares a function to be a *GPU kernel*. Recall `export` in ISPC → similar!
- `__device__`: declares a function that executes on the GPU and *is only callable on the GPU*.
- `__host__`: declares a function that executes on the host and *is callable on the host only*.
- The `__device__` and `__host__` specifiers can be used together. Two object files will be generated in this case.
- A `__global__` function must have a `void` return type and cannot be a member of a class.
- A call to a `__global__` function is *asynchronous* (it returns before the GPU has completed the task) and the call must specify its *execution configuration*.

NVIDIA CUDA: VECTOR ADDITION

Calling a GPU kernel from host code:

C++ main function:

```
1  int main(void)
2  {
3      int N = 1 << 20; // = 4096 * 256
4      float *x, *y;
5
6      // Unified Memory: accessible from CPU or GPU
7      cudaMallocManaged(&x, N * sizeof(float));
8      cudaMallocManaged(&y, N * sizeof(float));
9
10     // initialize x and y arrays on the host
11     for (int i = 0; i < N; i++) {
12         x[i] = 1.0f;
13         y[i] = 2.0f;
14     }
15
16     // Run kernel on 1M elements on the GPU (1 thread)
17     add<<<1, 1>>>(N, x, y);
18
19     // Wait for GPU to finish.  CUDA kernel calls are
20     // asynchronous and do not block the calling CPU
21     // thread.
22     cudaDeviceSynchronize();
23
24     // Free memory
25     cudaFree(x);
26     cudaFree(y);
27
28     return 0;
29 }
```

Steps:

- Allocate unified memory (accessible from host or device).
- Initialize the data *on the host* (accessing unified memory).
- Launch the GPU kernel. A kernel call has the following general *execution configuration* (often only the first two parameter within the triple-chevrons are required):

```
f<<<GridDim, BlockDim, ShMem, Stream>>>(param list);
```

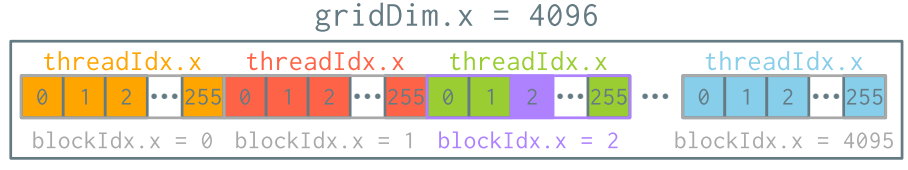
- Synchronize the GPU with the caller on the host (kernel calls are *asynchronous*).
- Free the allocated resources.

NVIDIA CUDA: VECTOR ADDITION

GPU kernel execution configuration:

- CUDA kernels are launched on a *grid of thread blocks*.
- In each kernel you have access to the execution configuration with *built-in variables* (similar to `programIndex` and `programCount` in ISPC):
  - `threadIdx`: the thread index within a thread block.
  - `blockIdx`: the block index in the kernel launch grid.
  - `blockDim`: the thread block dimensions (number of threads in each dimension).
  - `gridDim`: the dimension of the launch grid (number of thread blocks in each dimension).
- Each of these built-in variables are 3D. *For example:* `threadIdx.x`, `threadIdx.y` and `threadIdx.z` → CUDA supports 3D grids at most.

1D kernel grid:



`tid = blockIdx.x * blockDim.x + threadIdx.x`

Example: `tid = 2 * 256 + 2 = 514`

```
1  int main(void)
2  {
3      int N = 1 << 20; // = 4096 * 256
4      // skipping other code
5
6      // Run kernel on 1M elements on the GPU (with TLP)
7      int block_size = 256;
8      int n_blocks = (N + block_size - 1) / block_size;
9      add<<<n_blocks, block_size>>>(N, x, y);
10
11     // skipping other code
12     return 0;
13 }
```

Does our add kernel expose DLP (via TLP)?
It does not! This code so far ***is not parallel.***

NVIDIA CUDA: VECTOR ADDITION

Each CUDA thread runs the GPU kernel code!

Kernel launch examples:

```
1 add<<<1, 1>>>(N, x, y); // only 1 thread (no TLP)
2 add<<<1, 256>>>(N, x, y); // 256 threads (some TLP)
3
4 int block_size = 256; // cannot be arbitrarily large!
5 int n_blocks = (N + block_size - 1) / block_size;
6 add<<<n_blocks, block_size>>>(N, x, y); // lots of TLP!
```

GPU kernel must expose DLP (and ILP):

```
1 __global__ void add(int n, float *x, float *y)
2 {
3     // each thread executes the loop! No DLP!
4     for (int i = 0; i < n; i++)
5         y[i] = x[i] + y[i];
6 }
```

DLP aware CUDA kernel:

```
1 __global__ void add(int n, float *x, float *y)
2 {
3     // my thread ID in the grid
4     int tid = blockIdx.x * blockDim.x + threadIdx.x;
5     // total threads in execution configuration
6     int stride = blockDim.x * gridDim.x;
7     for (int i = tid; i < n; i += stride)
8         y[i] = x[i] + y[i];
9 }
```

This CUDA kernel is correct and exposes DLP via TLP (because each thread in the launch grid calls this kernel with its own `tid`). The kernel *does not* expose any ILP.

- `tid` is the index of the thread in the launch grid.
- `stride` contains the number of threads in the grid.
- *How many iterations per thread if there are exactly n threads in the launch grid?*
- *Will this kernel execute correctly if launched on a grid with fewer/more than n threads?*

NVIDIA CUDA: VECTOR ADDITION

Alternative way to write the kernel:

Loop-strided version:

```
__global__ void add(int n, float *x, float *y)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = tid; i < n; i += stride)
        y[i] = x[i] + y[i];
}
```

- Independent of execution configuration.
- Prefer to write CUDA kernels independent of launch grid.

Launch grid dependent version:

```
__global__ void add(int n, float *x, float *y)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < n) {
        y[tid] = x[tid] + y[tid];
    }
}
```

- Each thread performs one operation on different data (DLP exposure).
- The execution configuration must involve *at least* `n` threads.
- This kernel will not compute the correct result if launched with fewer than `n` threads. ***What if `n` is larger than what the execution configuration can support?***
- ***Is the `if`-statement necessary?***

NVIDIA. CUDA VERSUS ISPC

- Both CUDA and ISPC are based on a SPMD programming model. The "Single-Program" part maps to *threads* in CUDA and *program instances* (SIMD-lanes) in ISPC.
- Since both are based on a similar programming paradigm, similarities in the programming languages exist:

CUDA	ISPC
`__global__`	`export`
`warpSize`	`programCount`
`threadIdx.x/.y/.z`	`programInstance`

- CUDA warps *are not* maximally converged (since Volta architecture). A call to `__syncwarp()` is required for warp-level synchronization.
- In CUDA, DLP and TLP are *combined* at runtime (the GPU requires both). ISPC focus is on DLP (SIMD) only with TLP *complementary* added by programmer.
- In CUDA (or on GPU in general) exposure of ILP is mostly delegated to the programmer since the core architecture is much simpler compared to a CPU.

WHAT WE DID NOT DISCUSS

- We have only looked at an introductory CUDA kernel example and discussed how to launch a GPU kernel from host code.
- There are resource limitations imposed on the execution configuration that we did not discuss (maximum number of threads per block, shared memory availability and register usage per thread).
- GPU has fast shared memory for communication of data (explicit caching). To avoid data races thread-level and warp-level *synchronization* must be used.
- Memory access and bank conflicts.
- The CUDA API is very large, we have not touched any of it.
- GPUs are connected via PCIe to the host. This imposes a new bottleneck. Depending on the application, where should the application data be stored: *on the host or on the device (GPU)?*
- Homogeneous/heterogeneous computing, GPUs for acceleration.

RECAP

- CPUs are optimized for latency while GPUs are optimized for throughput.
- That is, caches are the key concept in CPU architectures to hide latency, while high degree of parallelism is needed to hide latencies on GPUs (Little's Law).
- Latency is a *response time* that does not depend on the payload/workload of the item. *Examples include:* number of cycles to execute an instruction, time it takes for an elevator ride, time it takes to wash dishes in dish washer, time it takes to send a message from point A to B.
- ***Little's Law*** relates latency and throughput to degree of concurrency (which is related to parallelism).
- Nvidia's CUDA programming model is a SPMD paradigm (similar to ISPC) where you put yourself in the position of a single CUDA thread. Many CUDA threads execute your GPU kernel code *concurrently*. DLP and TLP are intertwined on a GPU. If DLP or TLP is *insufficient*, you must increase ILP in your kernel code at the cost of registers.

Further reading:

- Chapter 4 in *"Computer Architecture"*,
  J. Hennessy and D. Patterson, Morgan Kaufmann 2019
- CUDA documentation: https://docs.nvidia.com/cuda/
- J. D. Little *"A Proof for the Queuing Formula: L = λW"*, Operations Research, 1961