University of Zurich[UZH]

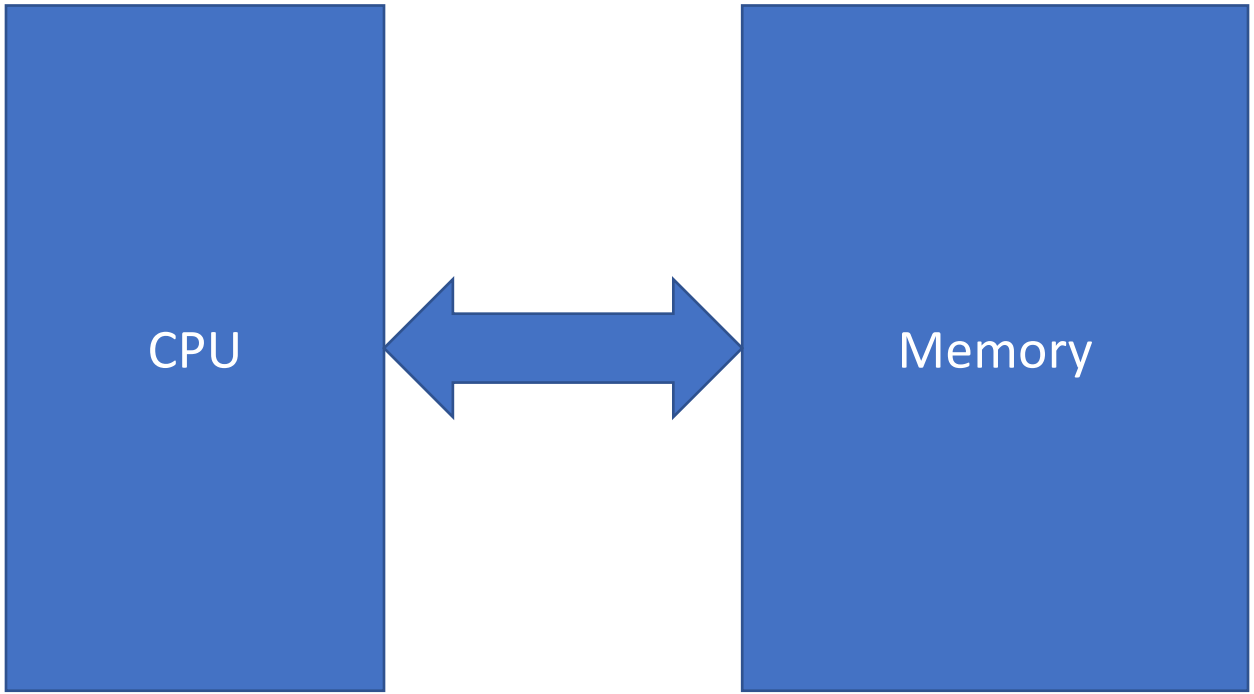
High Performance Computing Lecture 11

Douglas Potter
Jozef Bucko
Noah Kubli

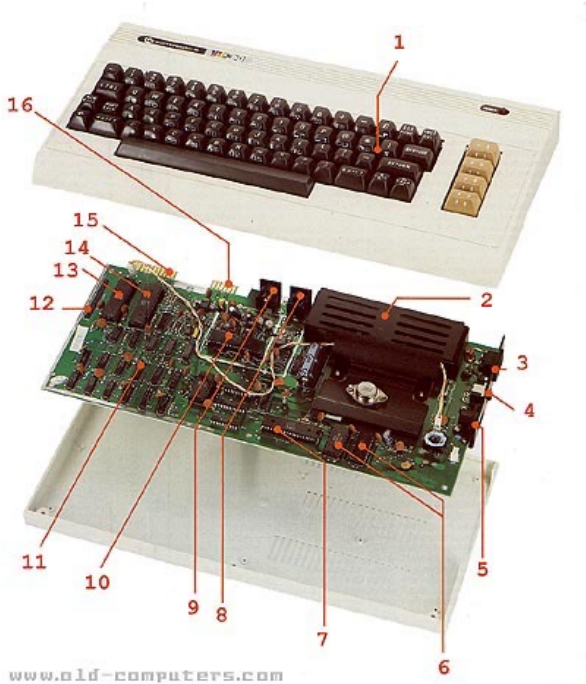
Hybrid Computing

Or what you already do…

Simple Model (1 CPU + Memory)



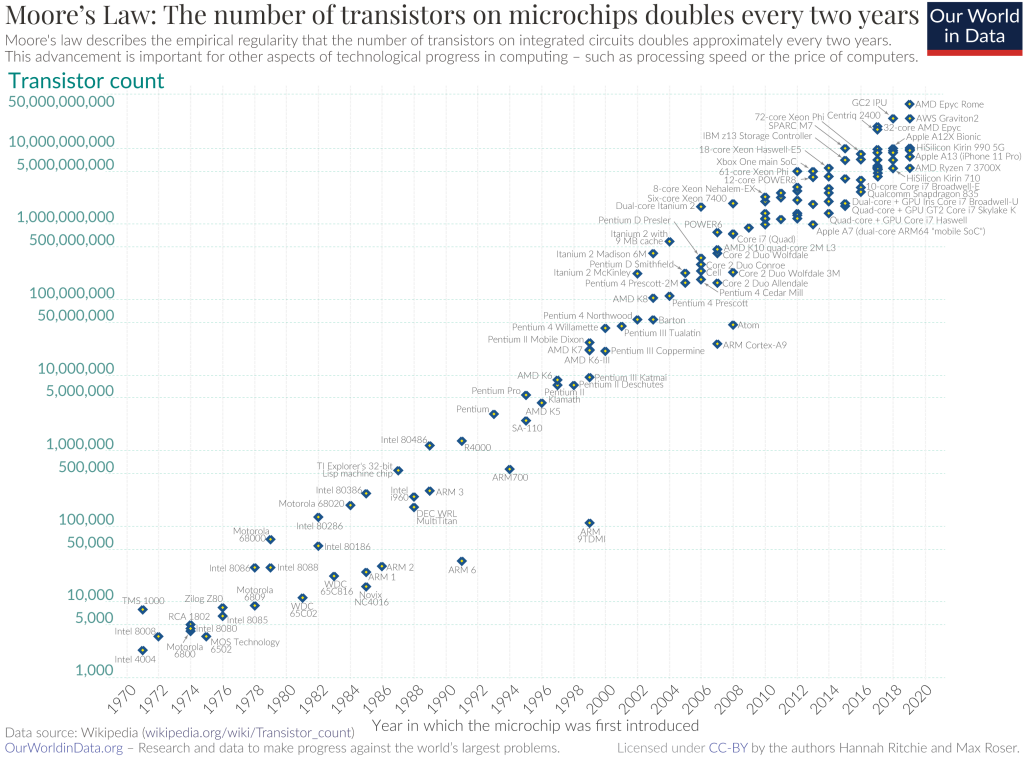
The Commodore VIC-20



1. Keyboard
2. ROM Expansion
3. Power connector
4. On/Off Switch
5. Joystick port
6. ROM chip with BASIC
7. 8-bit 6502 Microprocessor (1 MHz)
8. TV and RGB Input
9. Serial interface (e.g., floppy drive)
10. Video and Sound Chip (“VIC Chip”)
11. Static RAM (5 Kilobytes)
12. Keyboard connector
13. I/O Chip
14. I/O Chip
15. Serial mode connector (e.g., Modem)
16. Cassette tape connector (for saving programs)

Transistor Counts

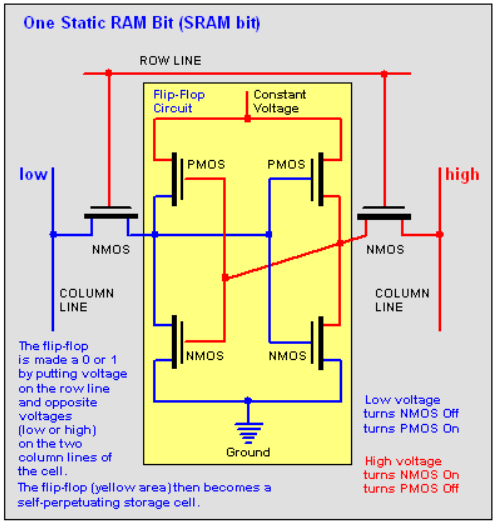
- Double every two years
- “Moore’s Law”
- 6502: 3510 transistors
- AMD EPYC: 40 billion!

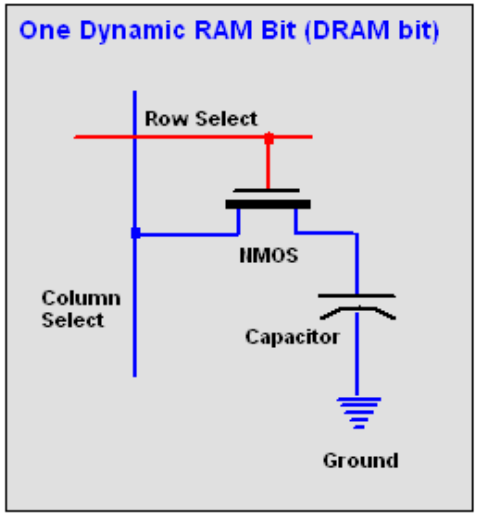


Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world's largest problems.

“Dynamic” versus “Static” RAM

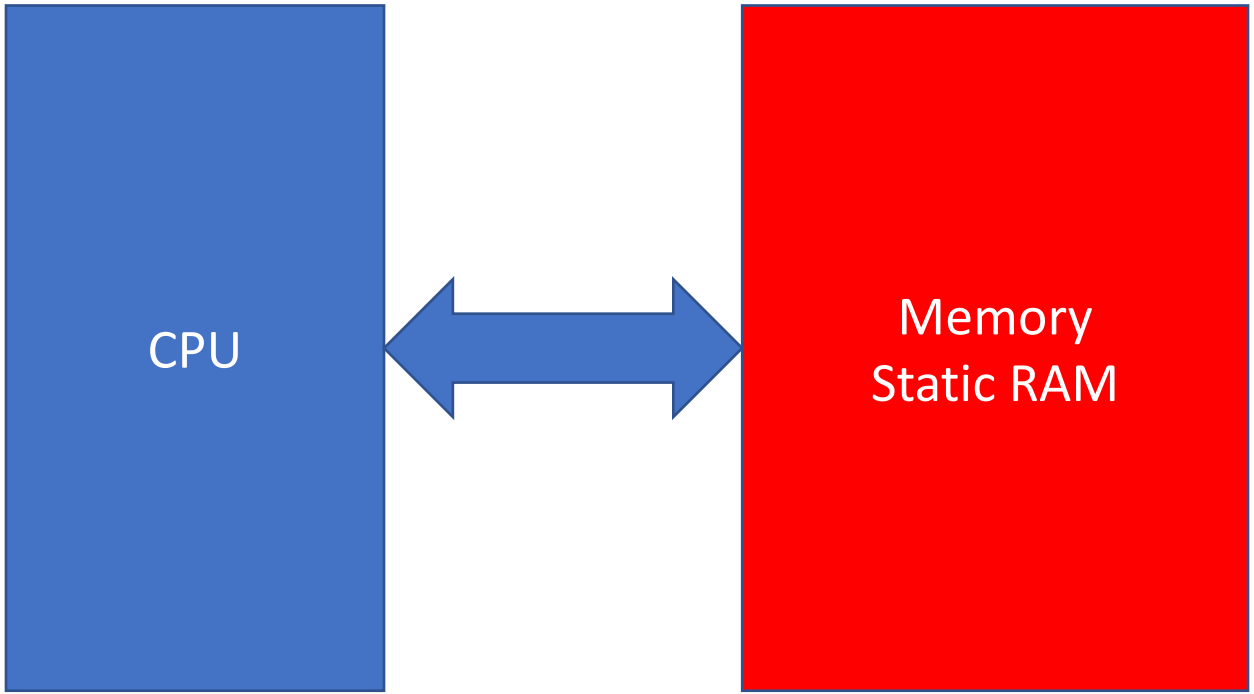
- Dynamic RAM (DRAM)
  - Single capacitor to hold each “bit”; one transistor
  - Capacitors leak, so it must be refreshed
  - Slow – high latency and “low” bandwidth
  - Large capacity and inexpensive
- Static RAM (SRAM)
  - Six transistors per “bit”
  - Does not need to be refreshed
  - Very fast
  - Expensive


One Static RAM Bit (SRAM bit)
ROW LINE
Flip-Flop Circuit
Constant Voltage
low
high
PMOS
PMOS
NMOS
NMOS
COLUMN LINE
COLUMN LINE
NMOS
NMOS
Ground
The flip-flop is made a 0 or 1 by putting voltage on the row line and opposite voltages (low or high) on the two column lines of the cell.
The flip-flop (yellow area) then becomes a self-perpetuating storage cell.
Low voltage turns NMOS Off turns PMOS On
High voltage turns NMOS On turns PMOS Off

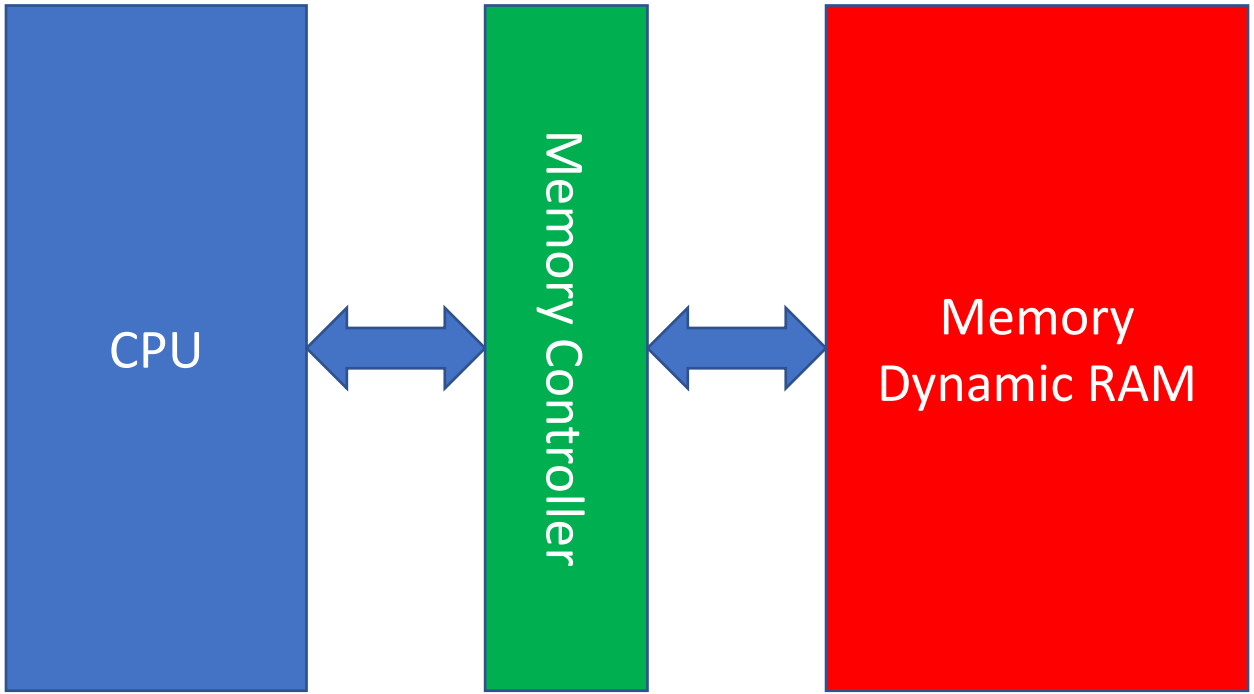

One Dynamic RAM Bit (DRAM bit)
Row Select
NMOS
Column Select
Capacitor
Ground

RAM Types

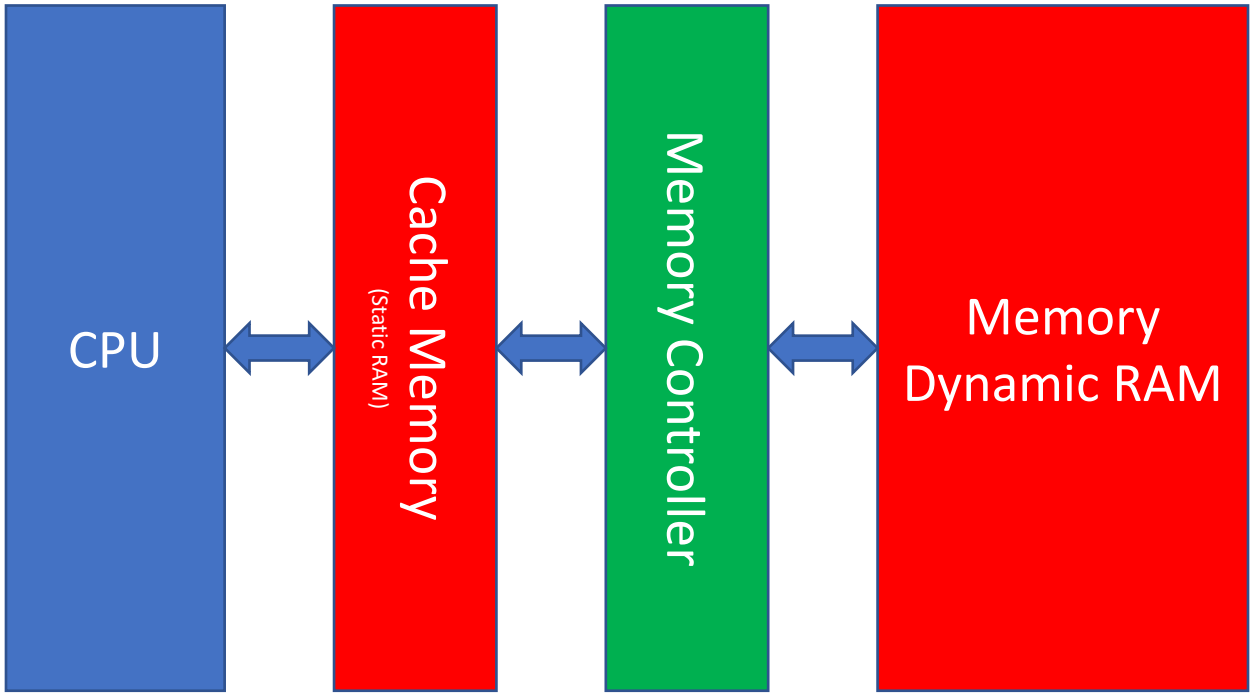
Simple Model (1 CPU + Memory)



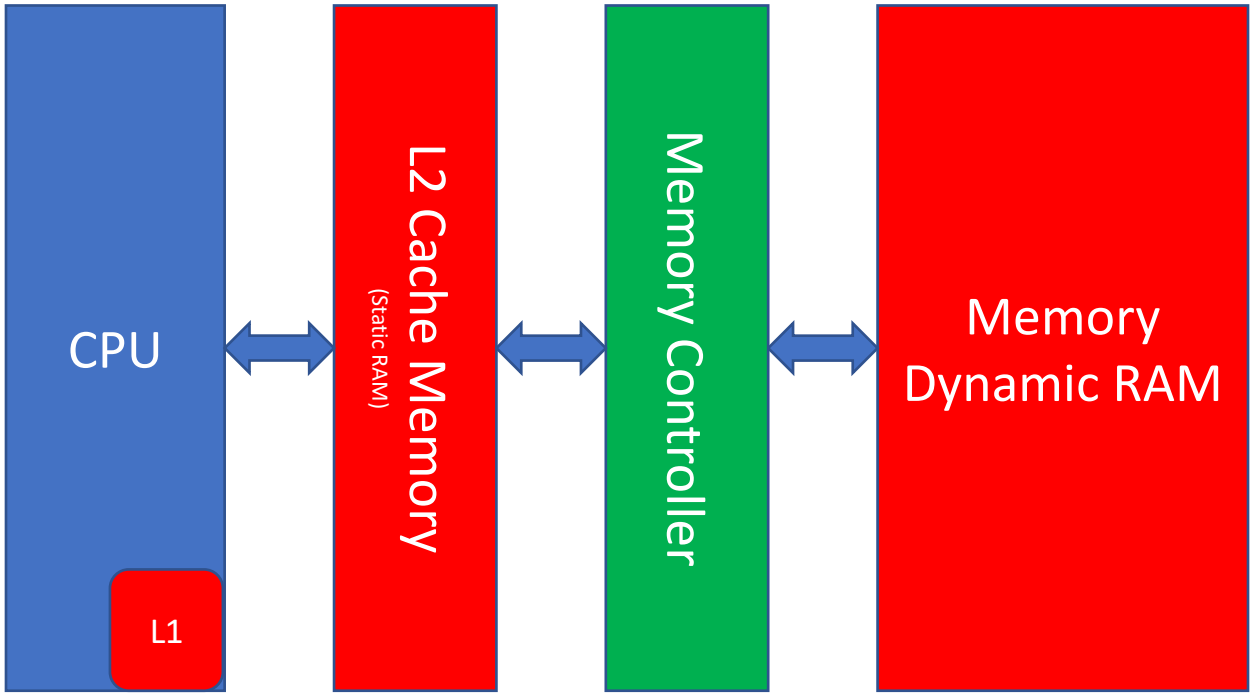
Simple Model (1 CPU + Memory)



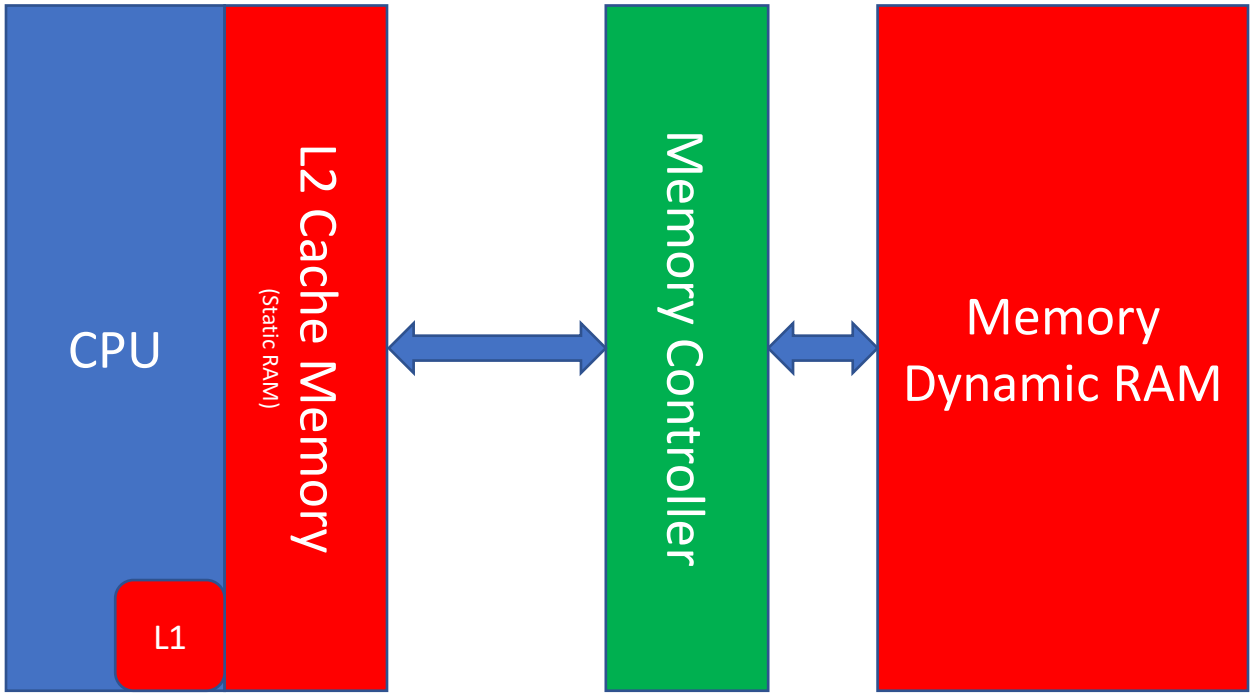
Add Cache


CPU
Cache Memory
(Static RAM)
Memory Controller
Memory
Dynamic RAM

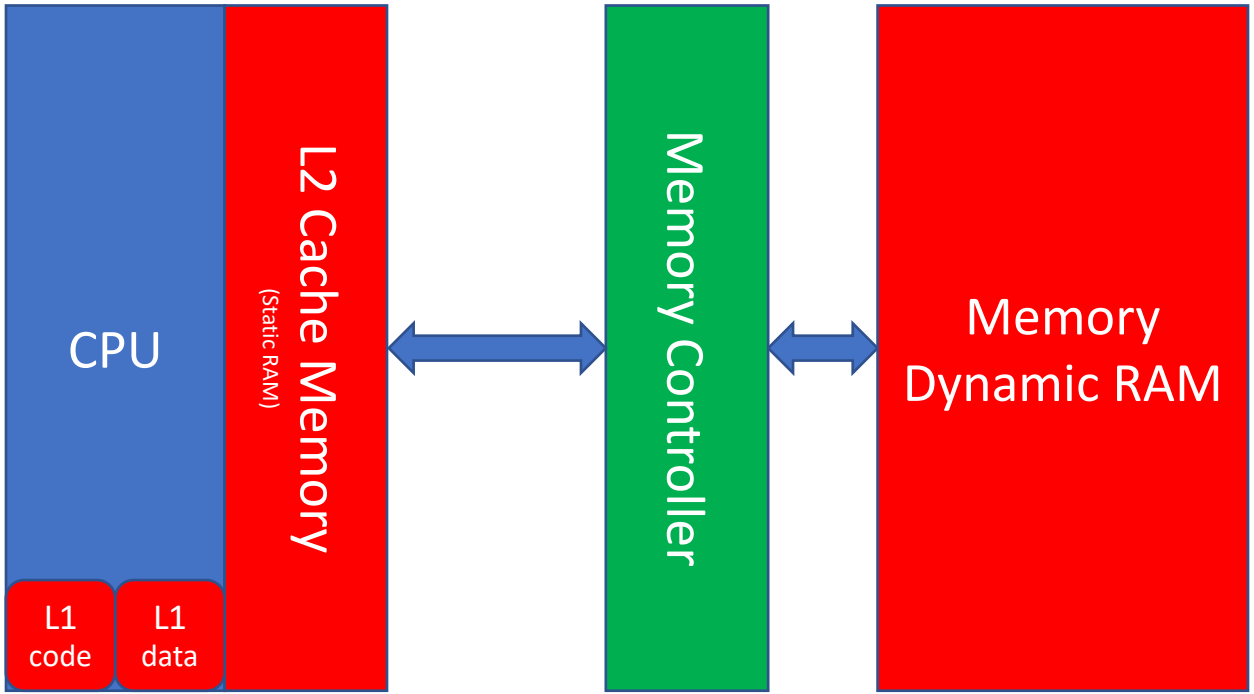
Multiple Levels of Cache



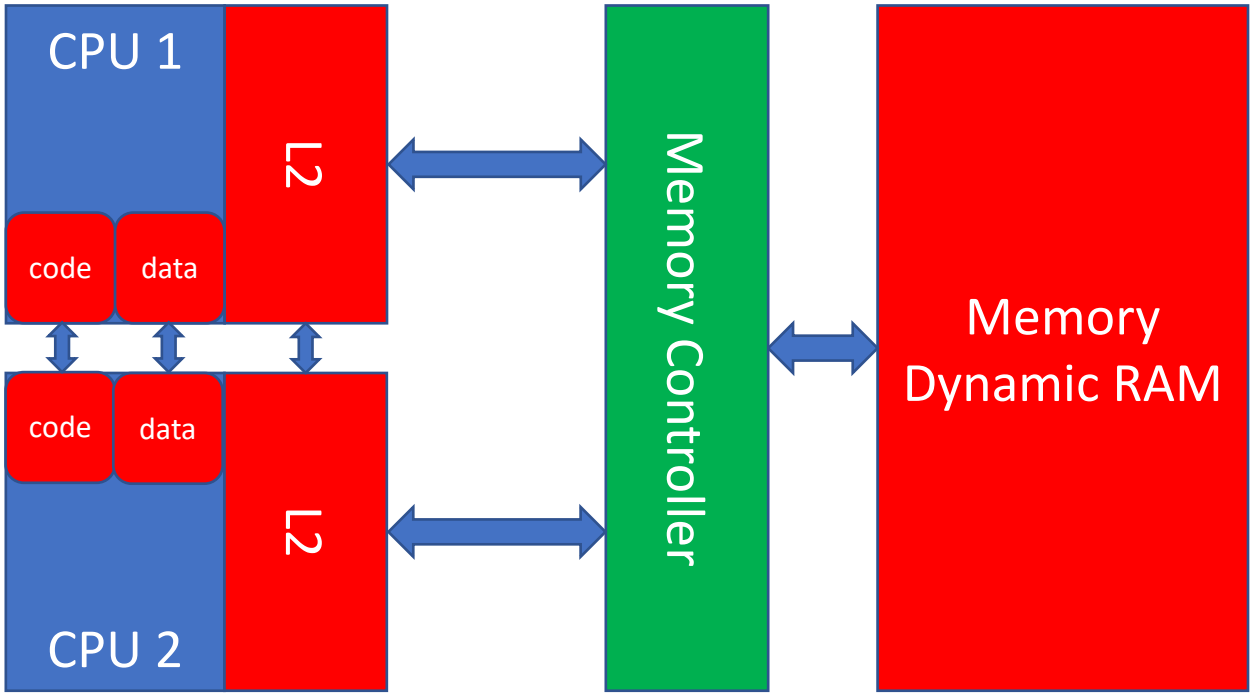
Multiple Levels of Cache



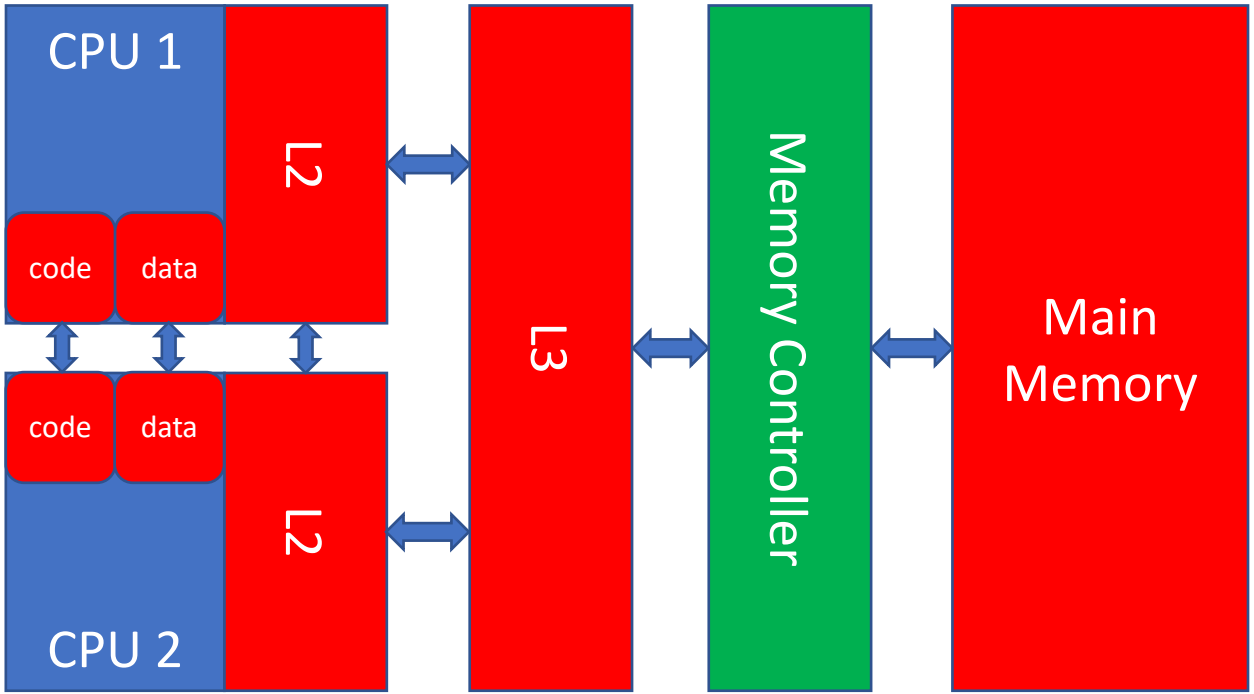
Multiple Levels of Cache



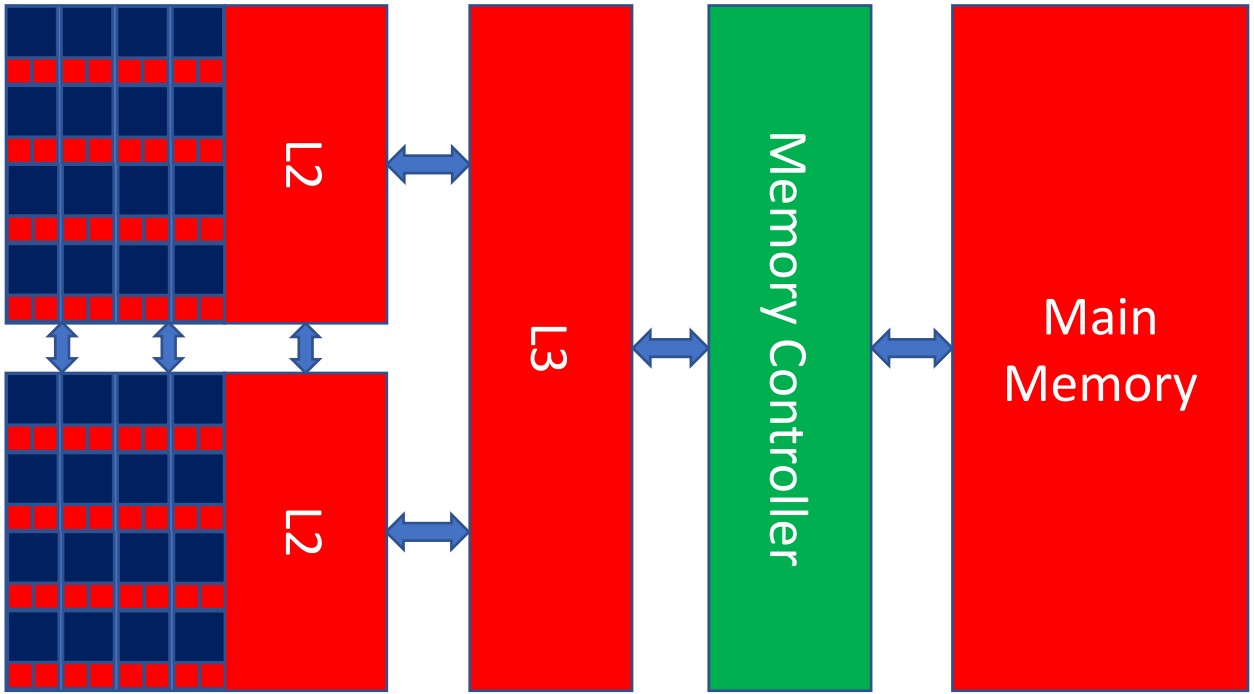
Add multiple CPUs



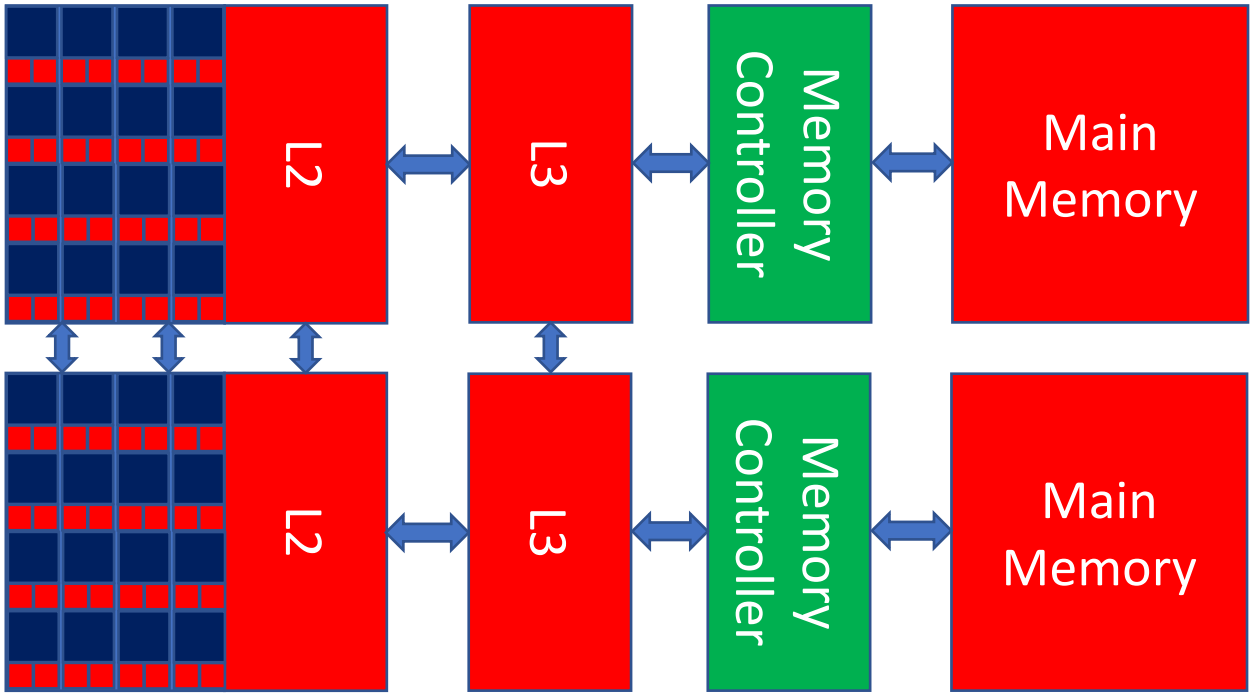
Even more Caches



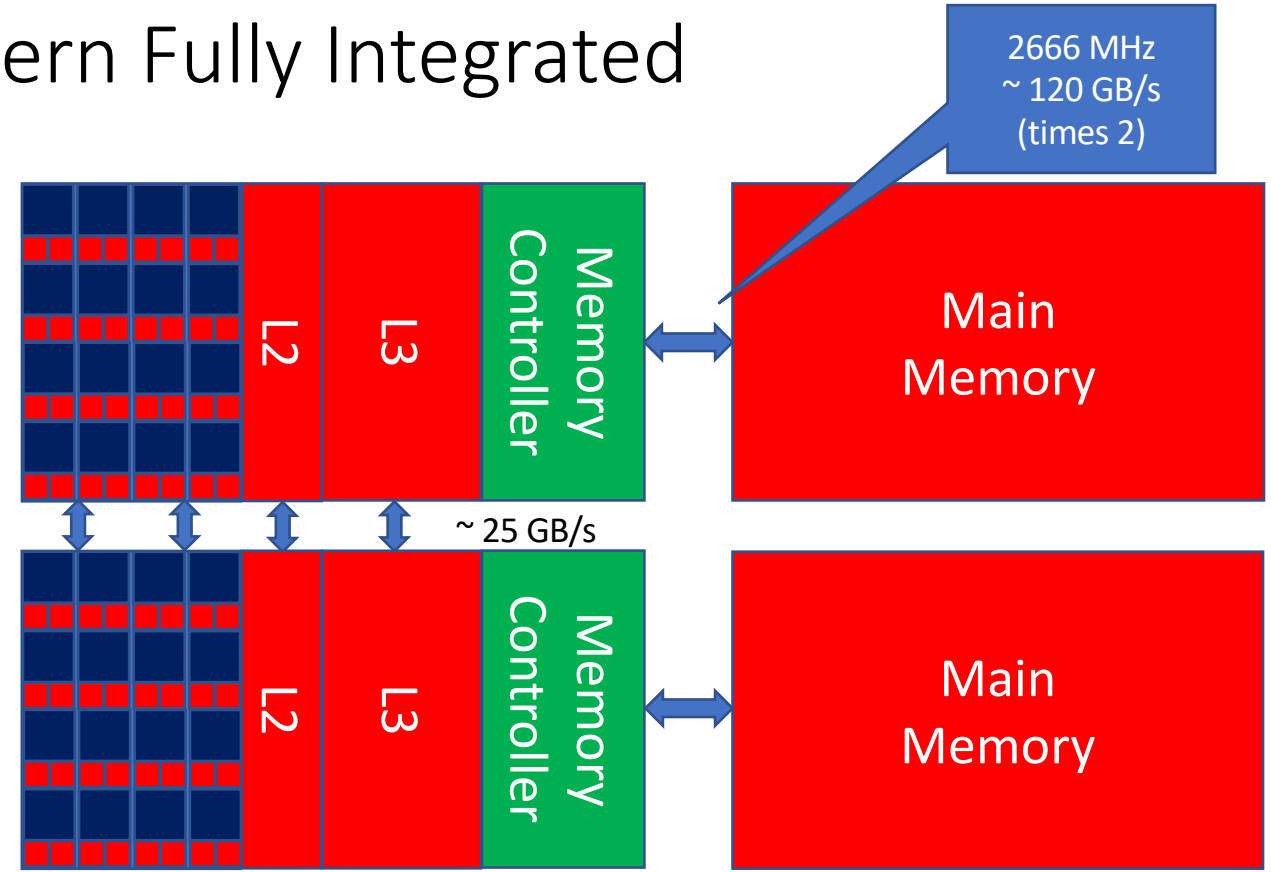
Add Cores to each CPU



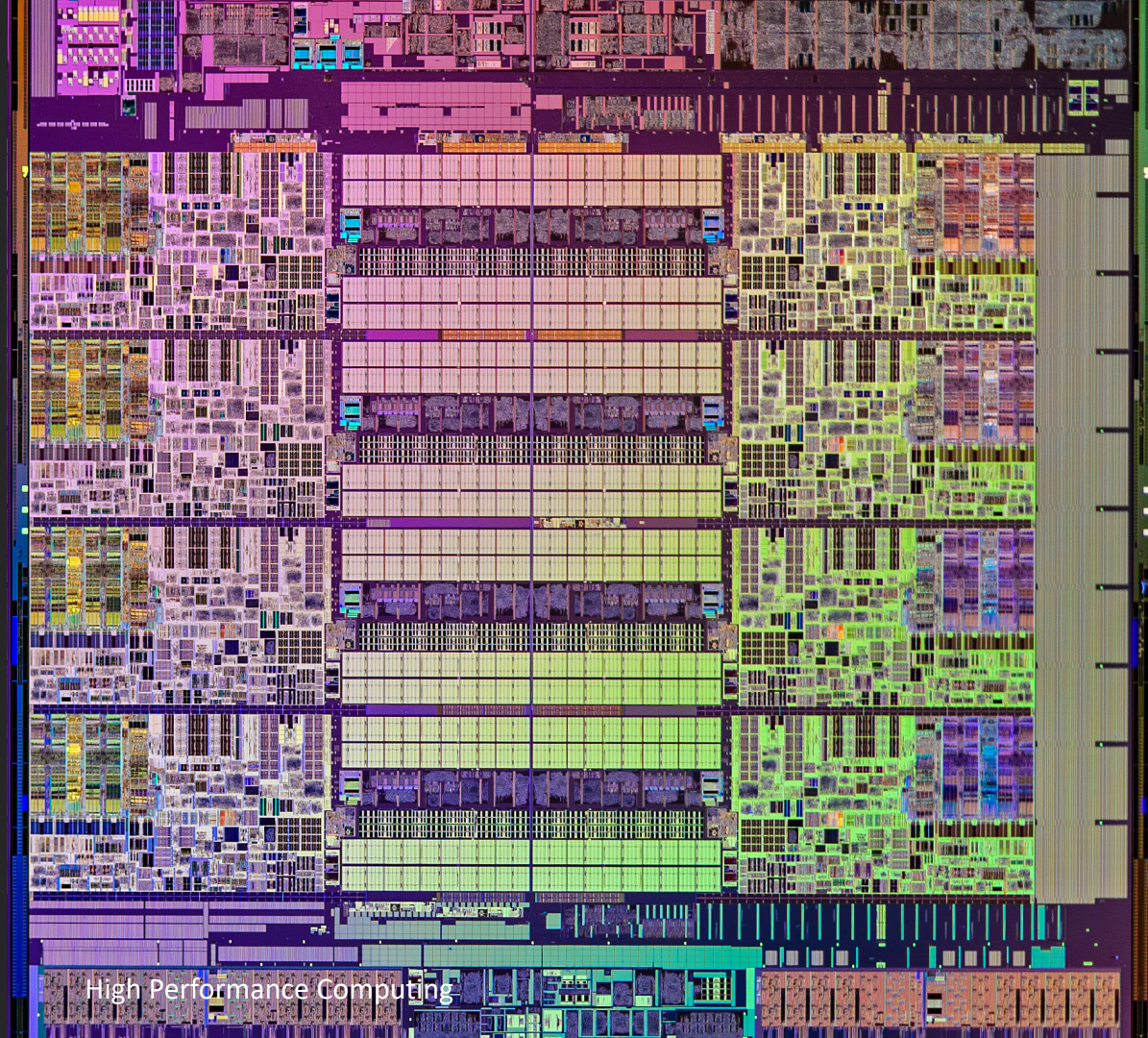
Non-Uniform Memory Access



Modern Fully Integrated

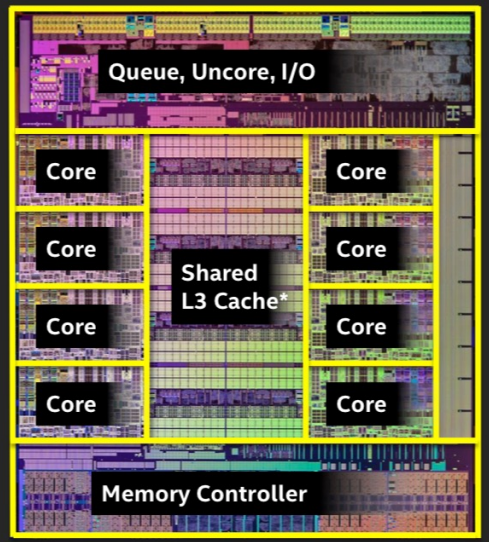


Xeon



Intel® Core™ i7-5960X Processor Die Map

22nm Tri-Gate 3-D Transistors



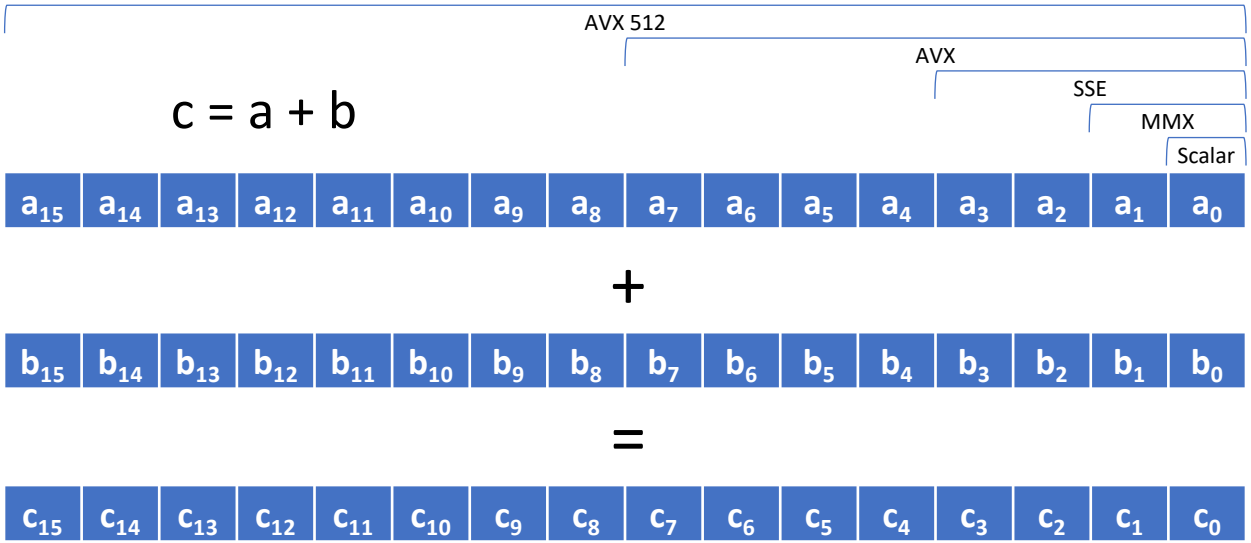
- Transistor count: 2.6 Billion
- Die size: 17.6mm x 20.2mm

*20MB of cache is
shared across all 8 cores

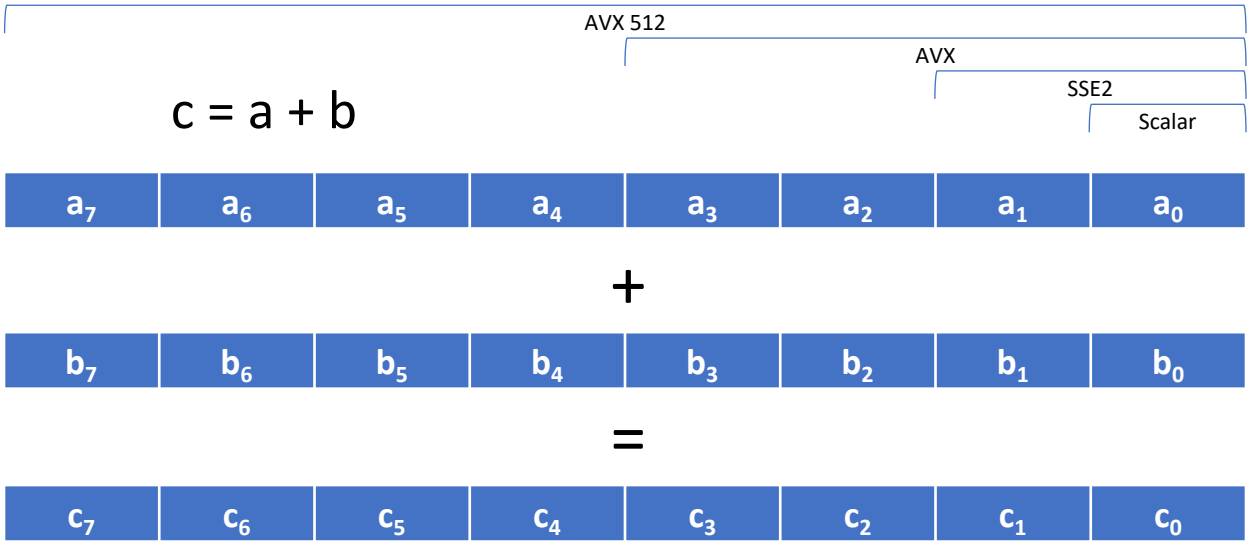
Under Embargo until
9:00am PST, August 29, 2014

intel Look Inside.

Vector Instructions (Single)



Vector Instructions (Double)



AVX Advantage – Xeon Gold 6126

Mode	Base GHz	Turbo Frequency (GHz) / Active Cores											
		1	2	3	4	5	6	7	8	9	10	11	12
Normal	2.6	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.3	3.3	3.3	3.3
AVX	2.2	3.6	3.6	3.4	3.4	3.3	3.3	3.3	3.3	2.9	2.9	2.9	2.9
AVX512	1.7	3.5	3.5	3.3	3.3	2.6	2.6	2.6	2.6	2.3	2.3	2.3	2.3

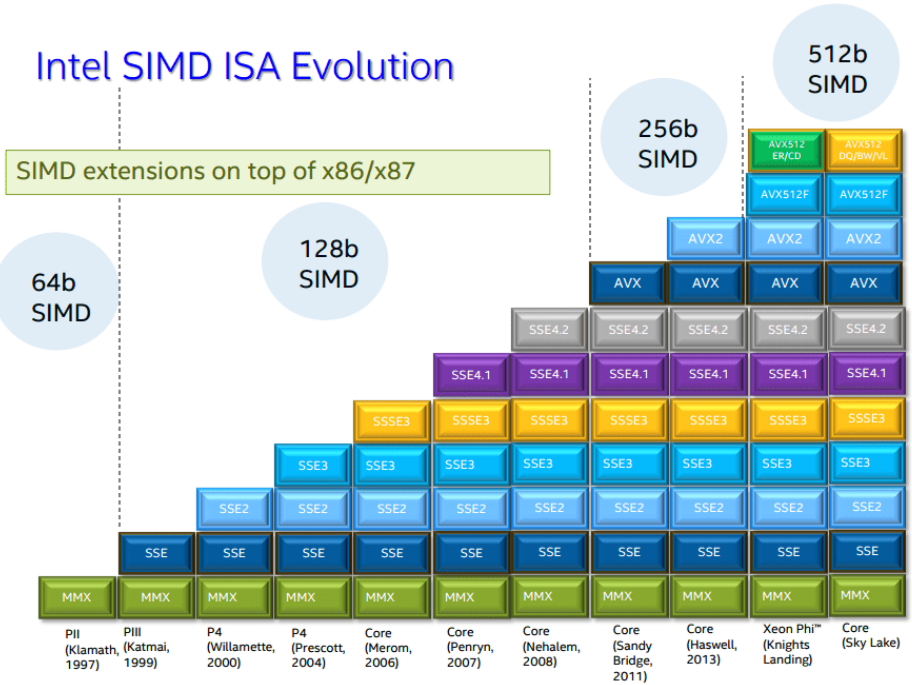
cores × speed × instructions per cycle × flops per instruction

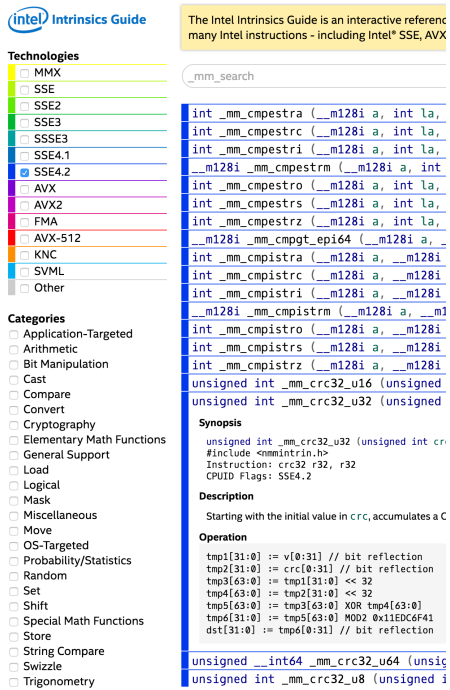
Single Core Scalar: $1 \times 3.7 \times 2 \times 1 = 7.4$ GFlops

Single Core AVX512: $1 \times 3.5 \times 2 \times 16 = 112$ GFlops

12 Core AVX512: $12 \times 2.3 \times 2 \times 16 = 883$ GFlops — 119 times faster

Perfect Scaling: $12 \times 3.7 \times 2 \times 16 = 1421$ GFlops — But 62% of perfect


Intel SIMD ISA Evolution
SIMD extensions on top of x86/x87
64b SIMD
128b SIMD
256b SIMD
512b SIMD
MMX
SSE
SSE2
SSE3
SSSE3
SSE4.1
SSE4.2
AVX
AVX2
AVX512F
AVX512 ER/CD
AVX512 DQ/BW/VL
PII (Klamath, 1997)
PIII (Katmai, 1999)
P4 (Willamette, 2000)
P4 (Prescott, 2004)
Core (Merom, 2006)
Core (Penryn, 2007)
Core (Nehalem, 2008)
Core (Sandy Bridge, 2011)
Core (Haswell, 2013)
Xeon Phi™ (Knights Landing)
Core (Sky Lake)


Intrinsics Guide
The Intel Intrinsics Guide is an interactive referenc
many Intel instructions - including Intel® SSE, AVX
Technologies
MMX
SSE
SSE2
SSE3
SSSE3
SSE4.1
SSE4.2
AVX
AVX2
FMA
AVX-512
KNC
SVML
Other
Categories
Application-Targeted
Arithmetic
Bit Manipulation
Cast
Compare
Convert
Cryptography
Elementary Math Functions
General Support
Load
Logical
Mask
Miscellaneous
Move
OS-Targeted
Probability/Statistics
Random
Set
Shift
Special Math Functions
Store
String Compare
Swizzle
Trigonometry
_mm_search
int _mm_cmpestra (__m128i a, int la,
int _mm_cmpestrc (__m128i a, int la,
int _mm_cmpestri (__m128i a, int la,
__m128i _mm_cmpestrm (__m128i a, int
int _mm_cmpestro (__m128i a, int la,
int _mm_cmpestrs (__m128i a, int la,
int _mm_cmpestrz (__m128i a, int la,
__m128i _mm_cmpgt_epi64 (__m128i a, _
int _mm_cmpistra (__m128i a, __m128i
int _mm_cmpistrc (__m128i a, __m128i
int _mm_cmpistri (__m128i a, __m128i
__m128i _mm_cmpistrm (__m128i a, __m1
int _mm_cmpistro (__m128i a, __m128i
int _mm_cmpistrs (__m128i a, __m128i
int _mm_cmpistrz (__m128i a, __m128i
unsigned int _mm_crc32_u16 (unsigned
unsigned int _mm_crc32_u32 (unsigned
Synopsis
unsigned int _mm_crc32_u32 (unsigned int cr
#include <nmmintrin.h>
Instruction: crc32 r32, r32
CPUID Flags: SSE4.2
Description
Starting with the initial value in crc, accumulates a C
Operation
tmp1[31:0] := v[0:31] // bit reflection
tmp2[31:0] := crc[0:31] // bit reflection
tmp3[63:0] := tmp1[31:0] << 32
tmp4[63:0] := tmp2[31:0] << 32
tmp5[63:0] := tmp3[63:0] XOR tmp4[63:0]
tmp6[31:0] := tmp5[63:0] MOD2 0x11EDC6F41
dst[31:0] := tmp6[0:31] // bit reflection
unsigned __int64 _mm_crc32_u64 (unsig
unsigned int _mm_crc32_u8 (unsigned i

Data Layout (Fortran)

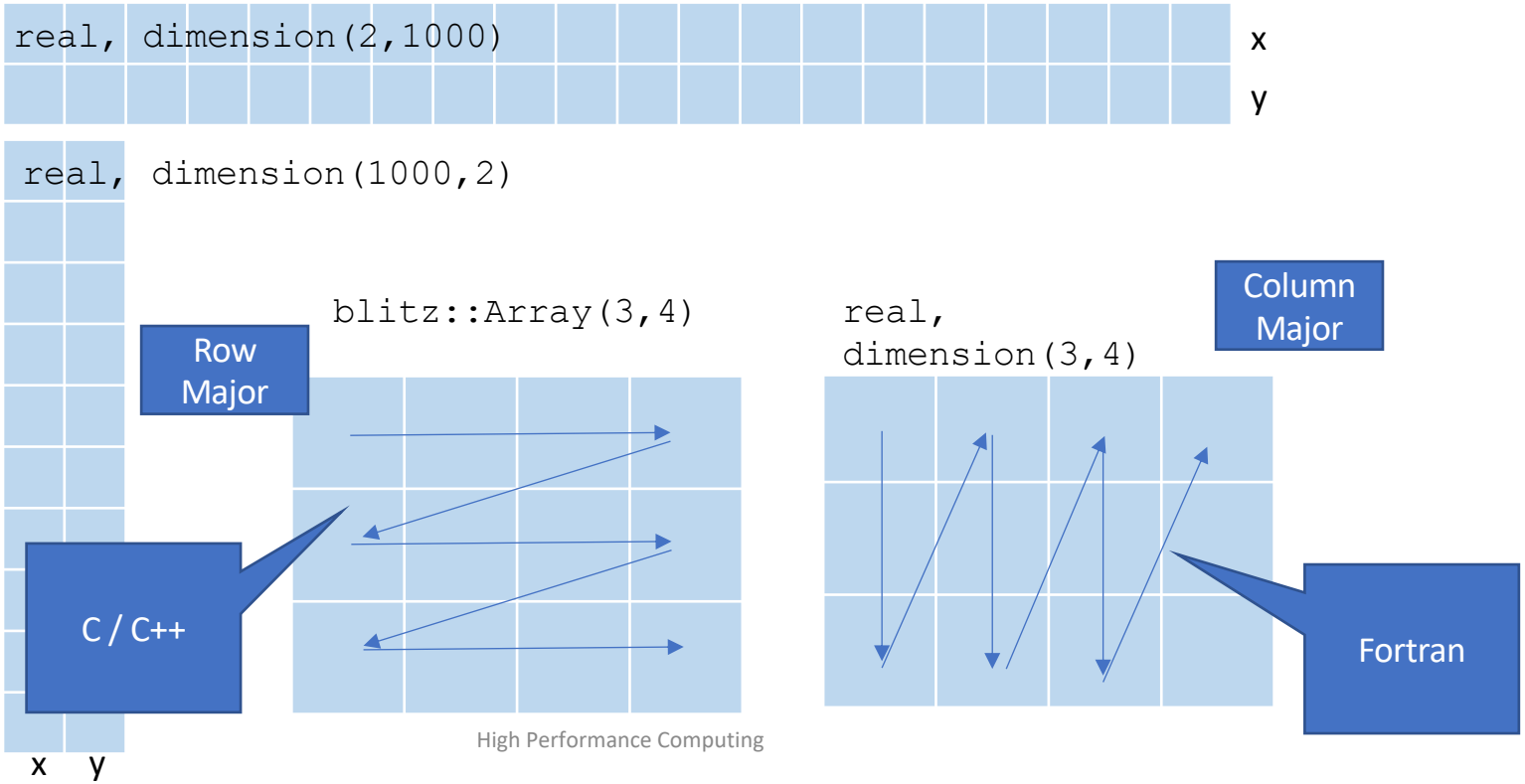
```
real, dimension(1000,2) :: r
real, dimension(1000) :: d2

do i = 1, 1000
  d2(i) = r(i,1)*r(i,1) + r(i,2)*r(i,2)
end do
```

$$d^2 = x^2 + y^2$$

Data Layout (Fortran)

```
real, dimension(2,1000) :: r
real, dimension(1000) :: d2

do i = 1, 1000
  d2(i) = r(1,i)*r(1,i) + r(2,i)*r(2,i)
end do
```

Data Layout (Fortran)



Does it Really Matter?

```
hpc:$ cat a1.f90
program array
  real,dimension(10000,10000) :: a
  integer :: i,j

  do i=1,10000
    do j=1,10000
      a(i,j) = i*7 + j*3
    end do
  end do
end program array
hpc:$ gfortran -O2 -o a1 a1.f90
hpc:$ time ./a1

real  0m1.827s
user  0m1.639s
sys   0m0.162s
```

Does it Really Matter?

```
hpc:$ cat a2.f90
program array
  real,dimension(10000,10000) :: a
  integer :: i,j

  do j=1,10000
    do i=1,10000
      a(i,j) = i*7 + j*3
    end do
  end do
end program array
hpc:$ gfortran -O2 -o a2 a2.f90
hpc:$ time ./a2

real  0m0.268s
user  0m0.084s
sys   0m0.185s
```

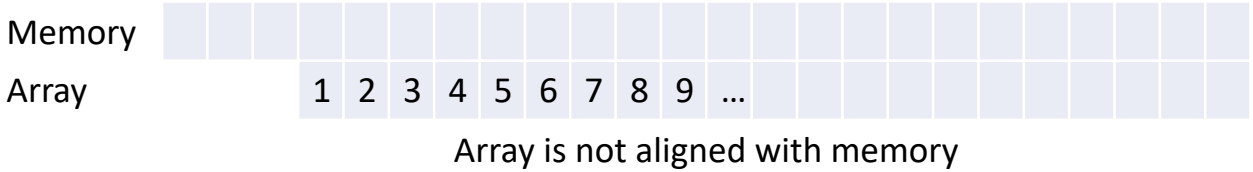
Some compiler figure it out!

```
hpc:$ ifort -O2 -o a1 a1.f90
hpc:$ ifort -O2 -o a2 a2.f90
hpc:$ time ./a1

real    0m0.184s
user    0m0.028s
sys     0m0.157s
hpc:$ time ./a2

real    0m0.175s
user    0m0.037s
sys     0m0.139s
```

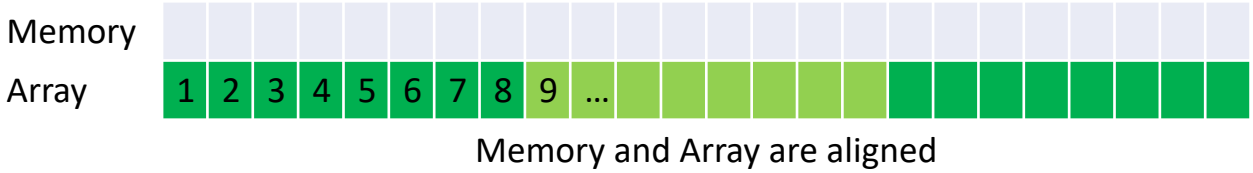
Data Layout – Separate Arrays

```
real, dimension(1000) :: x, y
real, dimension(1000) :: d2

do i = 1, 1000
  d2(i) = x(i)*x(i) + y(i)*y(i)
end do
```

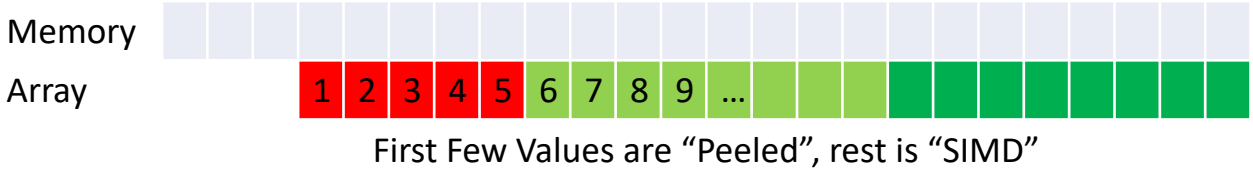
Alignment



Array is not aligned with memory



Memory and Array are aligned



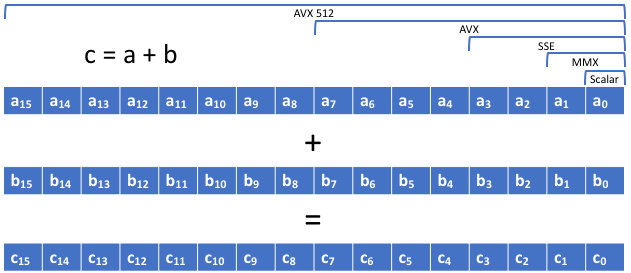
First Few Values are “Peeled”, rest is “SIMD”

Data Layout – Alignment hints

```
real, dimension(2,1000) :: r
real, dimension(1000) :: d2
!DIR$ ASSUME_ALIGNED r: 64
!DIR$ ASSUME_ALIGNED d2: 64

do i = 1, 1000
  d2(i) = r(1,i)*r(1,i) + r(2,i)*r(2,i)
end do
```

Divergence

```
real, dimension(1000) :: a, b, c
real :: v

do i = 1, 1000
  v = a(i) + b(i)
  if v .ge. 0 then
    v = sqrt(v)
  end if
  c(i) = v
end do
```

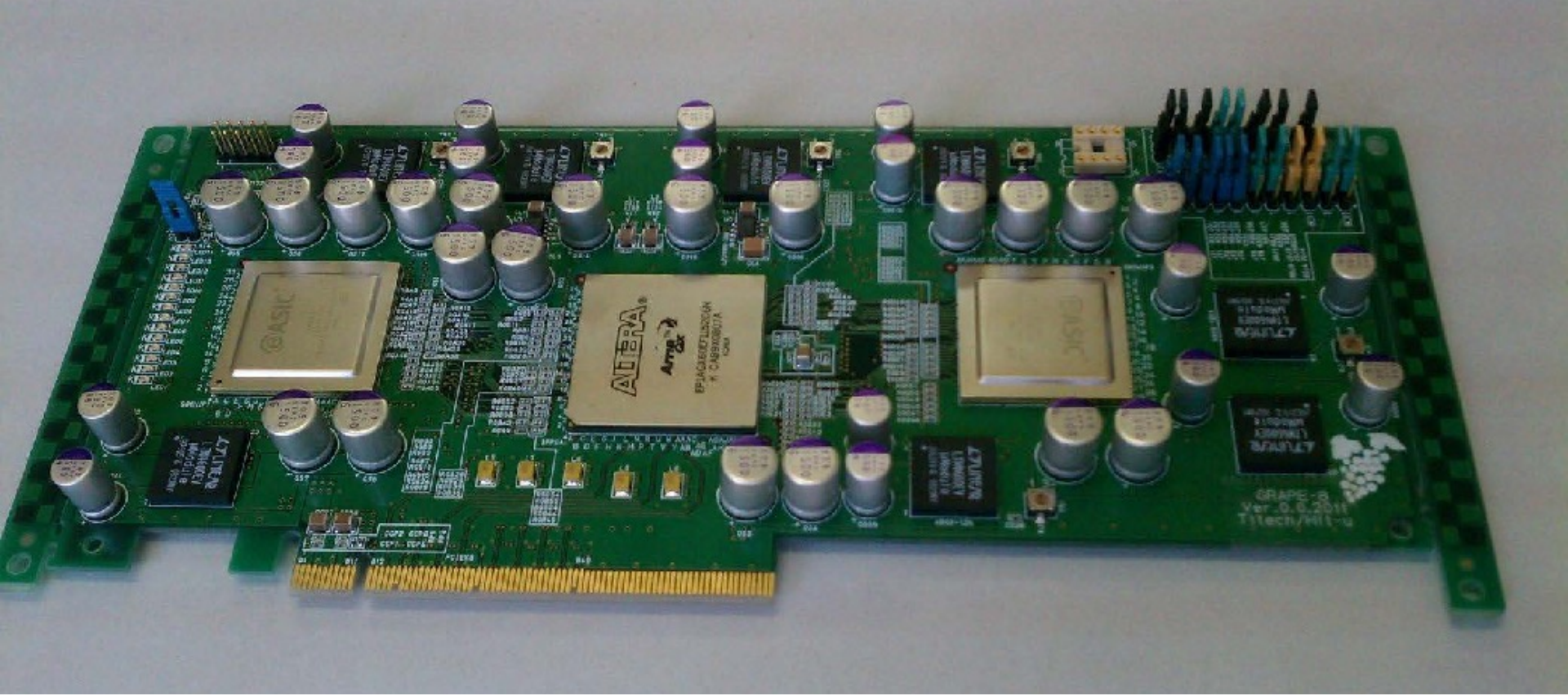


Summary (so far)

- When possible,
  - Keep data in cache (stay local)
  - Keep data in “vector” format
  - Keep data aligned
  - Access data in the “correct” order
  - Avoid divergence (or mitigate)

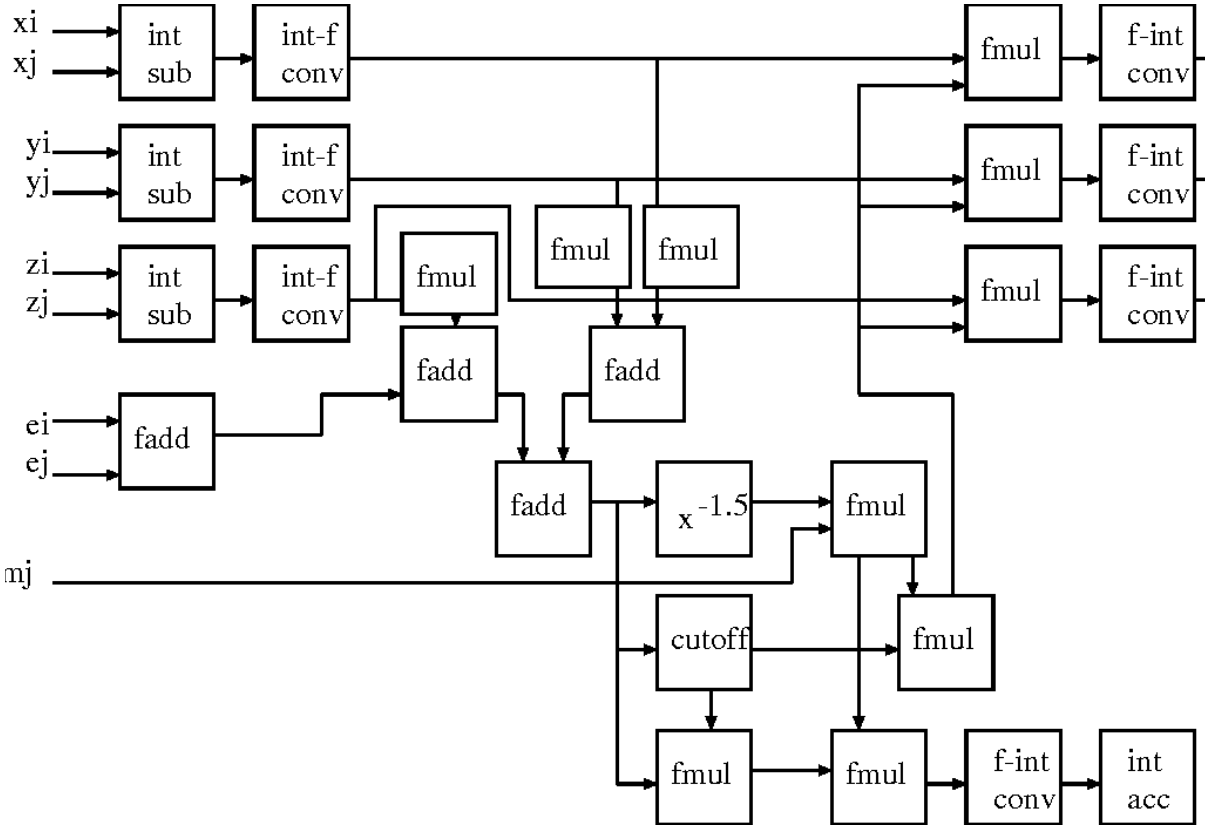
Other Architectures

History remembers them



GRAPE-8 – GRAvity PipE

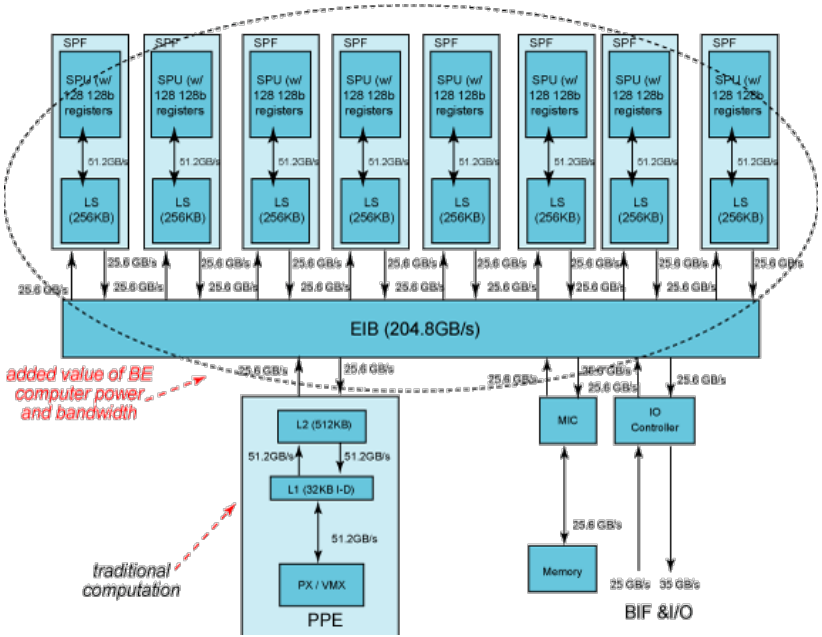
High Performance Computing

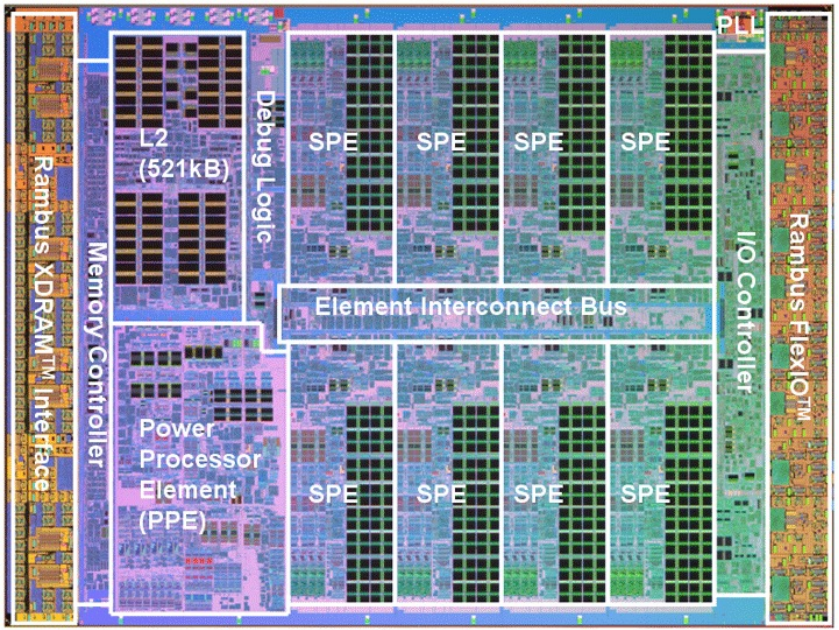


$$\Phi = -\frac{1}{\sqrt{r^2 + \epsilon^2}}$$

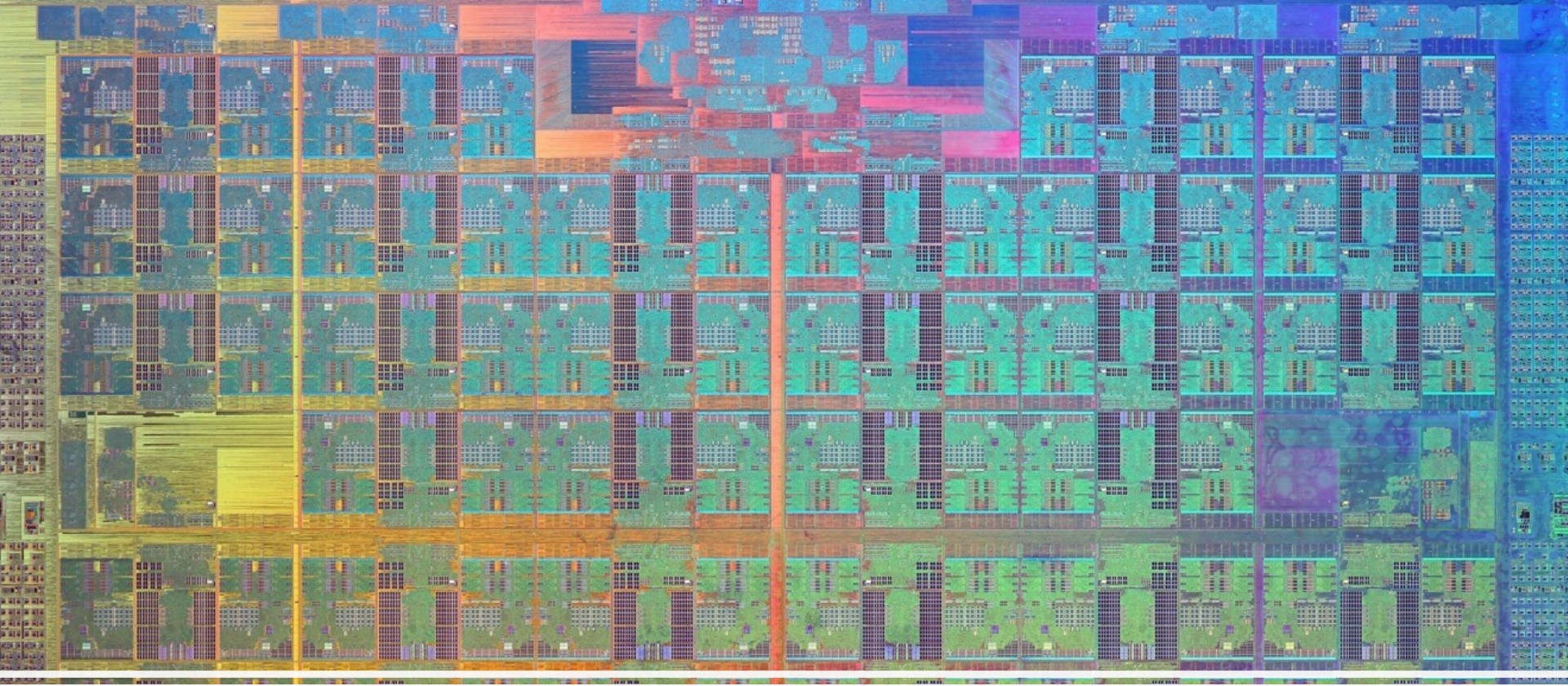
GRAPE-8 Block Diagram

$$\boldsymbol{a} = \frac{d\boldsymbol{v}_i}{dt} = \sum_{j \neq i}^{N} G m_j \frac{\boldsymbol{r}_j - \boldsymbol{r}_i}{|\boldsymbol{r}_j - \boldsymbol{r}_i|^3}$$



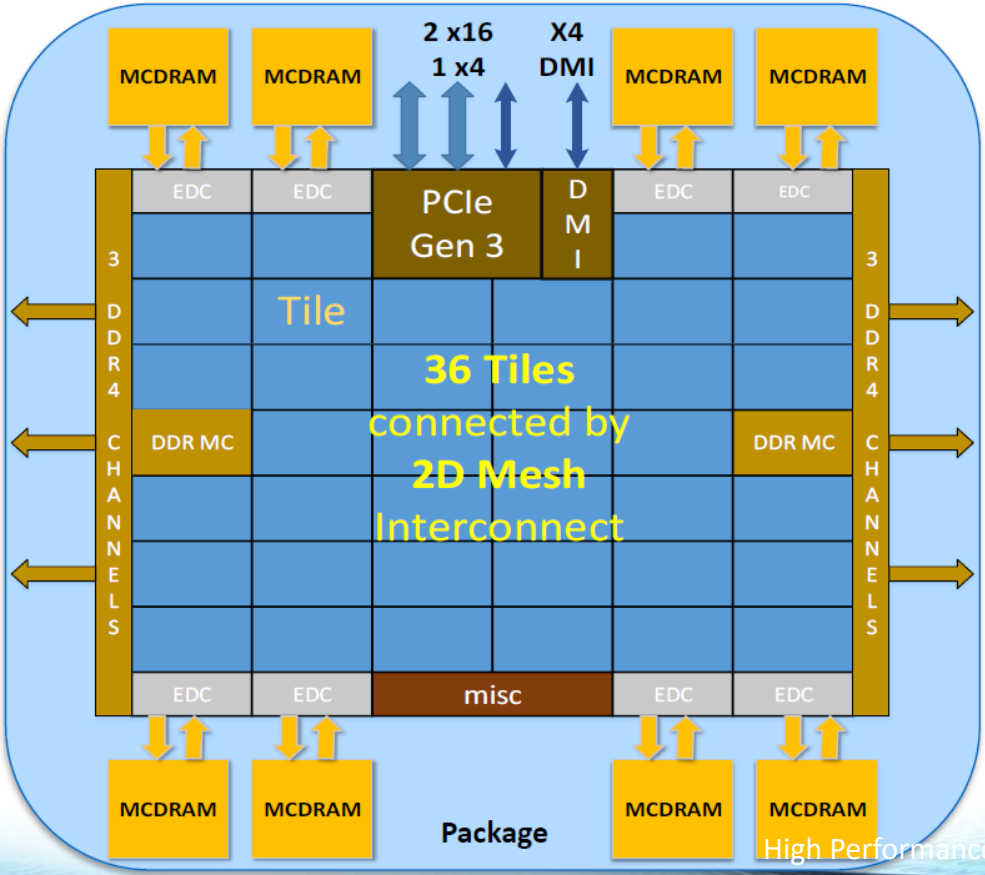


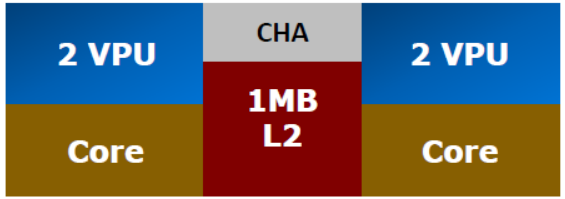
Cell Processor (Playstation 3)



Intel Xeon Phi

Knights Landing Overview





Chip: 36 Tiles interconnected by **2D Mesh**

Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW

DDR4: 6 channels @ 2400 up to 384GB

IO: 36 lanes PCIe Gen3. 4 lanes of DMI for chipset

Node: 1-Socket only

Fabric: Omni-Path on-package (not shown)

Vector Peak Perf: 3+TF DP and 6+TF SP Flops

Scalar Perf: ~3x over Knights Corner

Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

Source Intel: All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. KNL data are preliminary based on current expectations and are subject to change without notice. 1Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX). 2Bandwidth numbers are based on STREAM-like memory access pattern when MCDRAM used as flat memory. Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

Graphics Processing Unit

Mostly nVidia

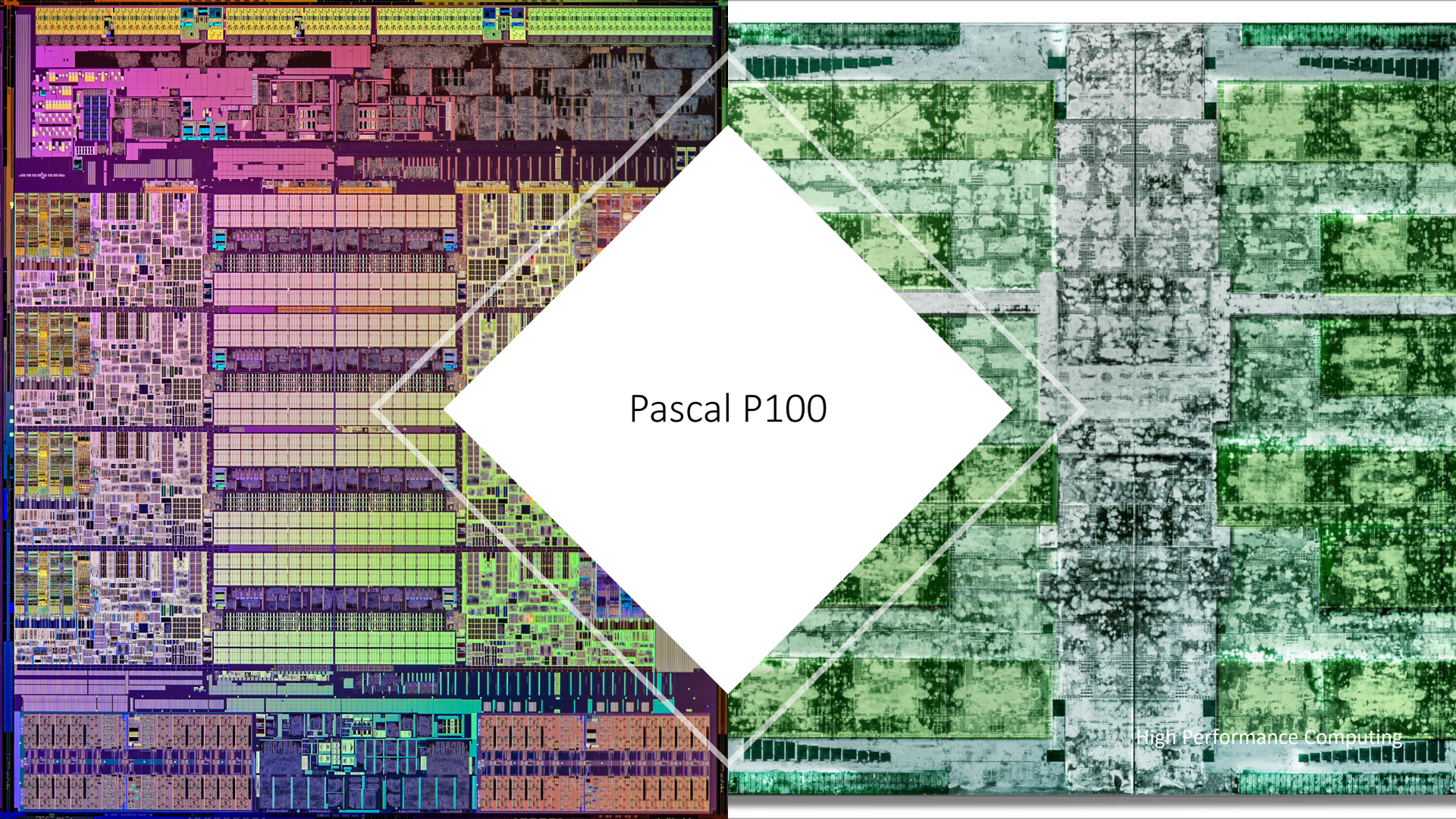
CPU versus GPU

CPU

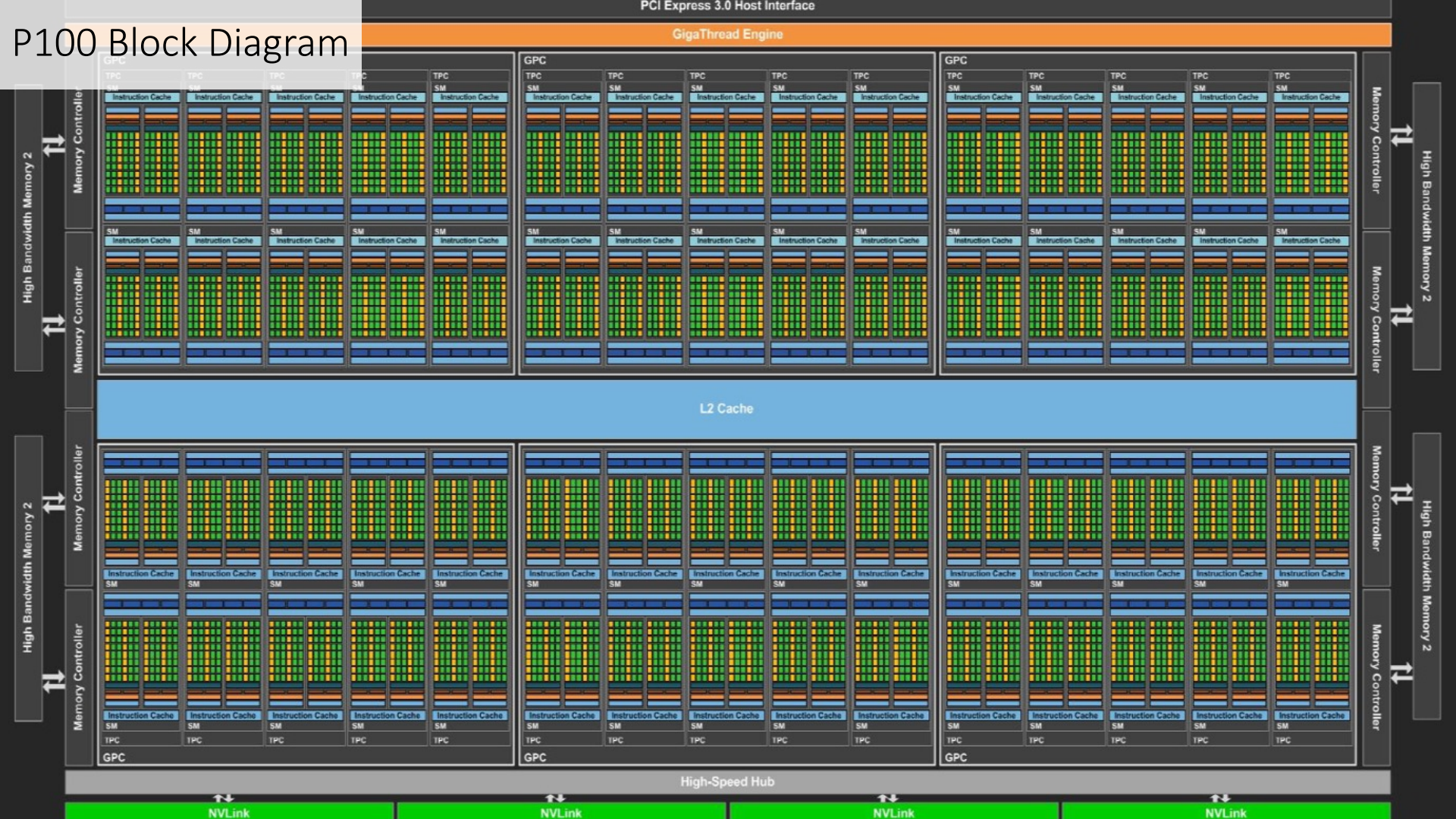
- General purpose
- Very large main memory
- Very fast clock speeds
- Low latency through caches
- Lower memory bandwidth
- Low performance per Watt

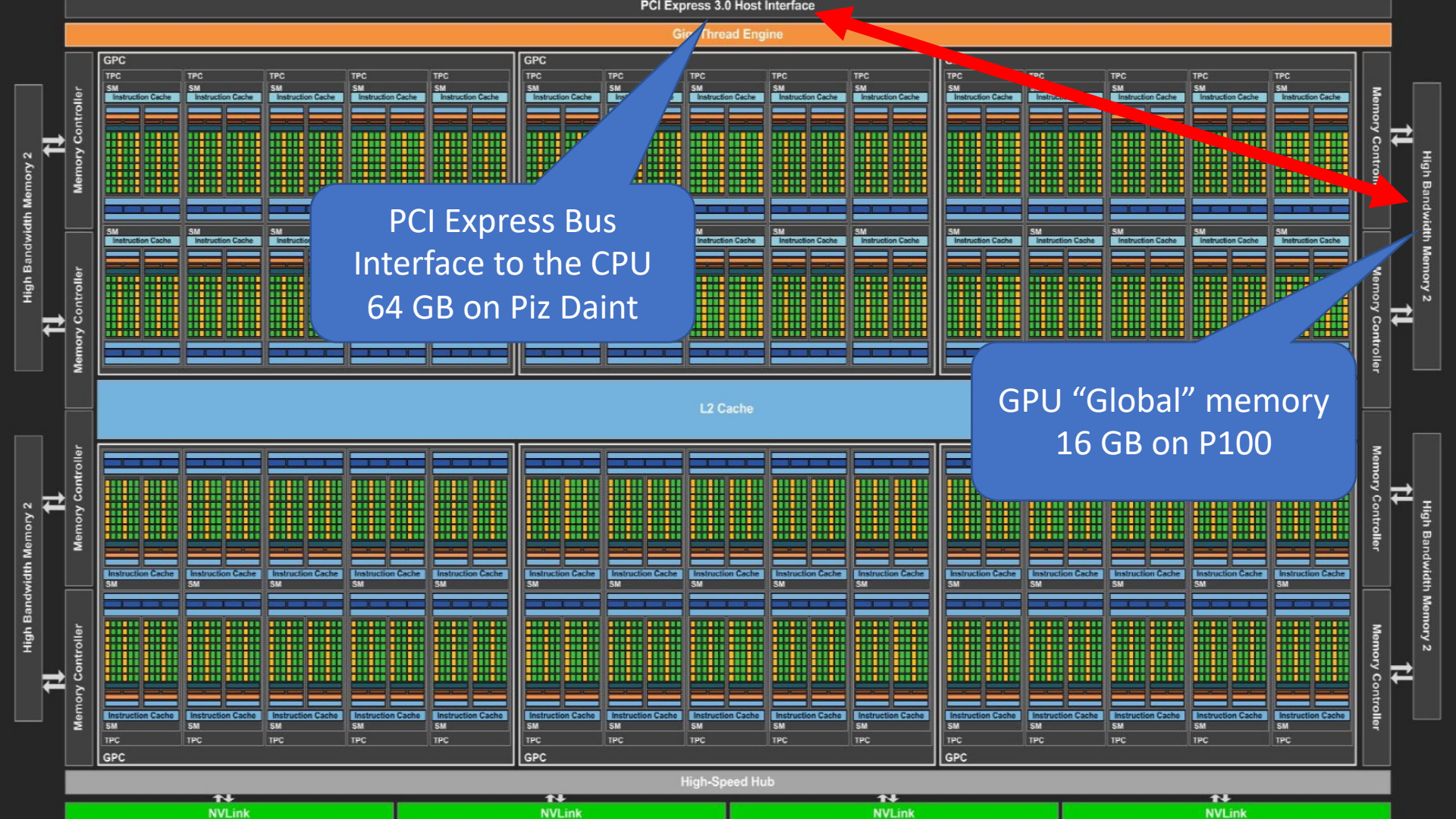
GPU

- Optimized for parallel tasks
- Lower main memory
- Low per-thread performance
- Latency hidden with threads
- More compute resources


Pascal P100
High Performance Computing

P100 Block Diagram




PCI Express 3.0 Host Interface
Giga Thread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub
NVLink
PCI Express Bus
Interface to the CPU
64 GB on Piz Daint
GPU “Global” memory
16 GB on P100

Host (CPU) to GPU Bottleneck

P100 (Piz Daint)

- PCIe 3.0, 32 GB/s
- 9300 Gflops (4 byte single)
- In one second
  - Transfer 8 billion floats to GPU
  - Perform 9300 billion flops
- 1100+ floating point operations for every float transferred
- Flops to bytes: ~ 300

A100 (More recent)

- PCIe 4.0, 64 GB/s
- 19500 Gflops (4 byte single)
- In one second
  - Transfer 16 billion floats to GPU
  - Perform 19500 billion flops
- 1200+ floating point operations for every float transferred
- Flops to bytes: ~ 300

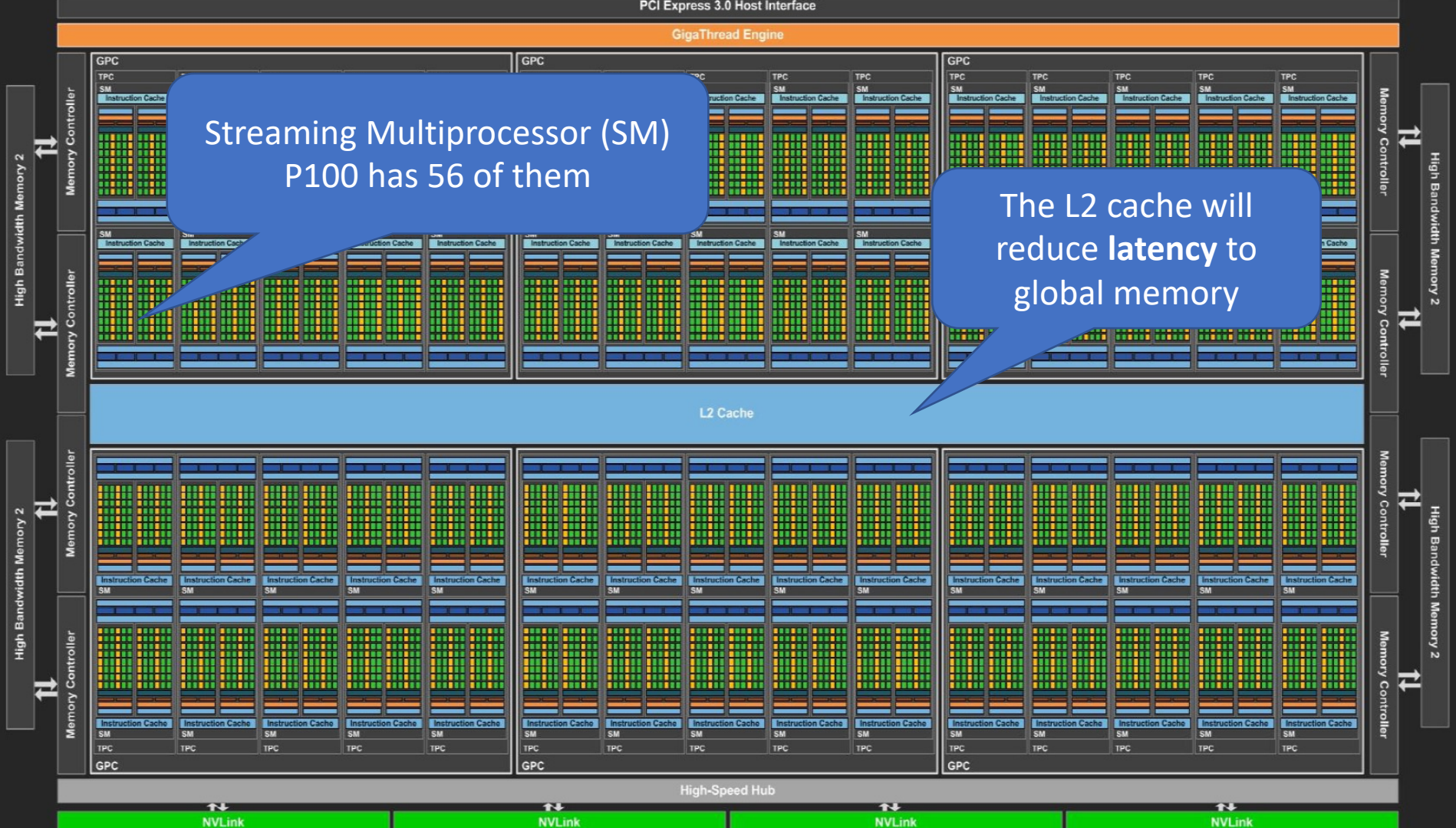
GPU Global memory bottleneck

P100

- Memory, 732 GB/s
- 9300 Gflops (4 byte single)
- In one second
  - read 183 billion floats
  - Perform 9300 billion flops
- 50 floating point operations for every float read from memory

A100

- Memory, 1555 GB/s
- 19500 Gflops (4 byte single)
- In one second
  - Transfer 388 billion floats
  - Perform 19500 billion flops
- 50 floating point operations for every float read from memory

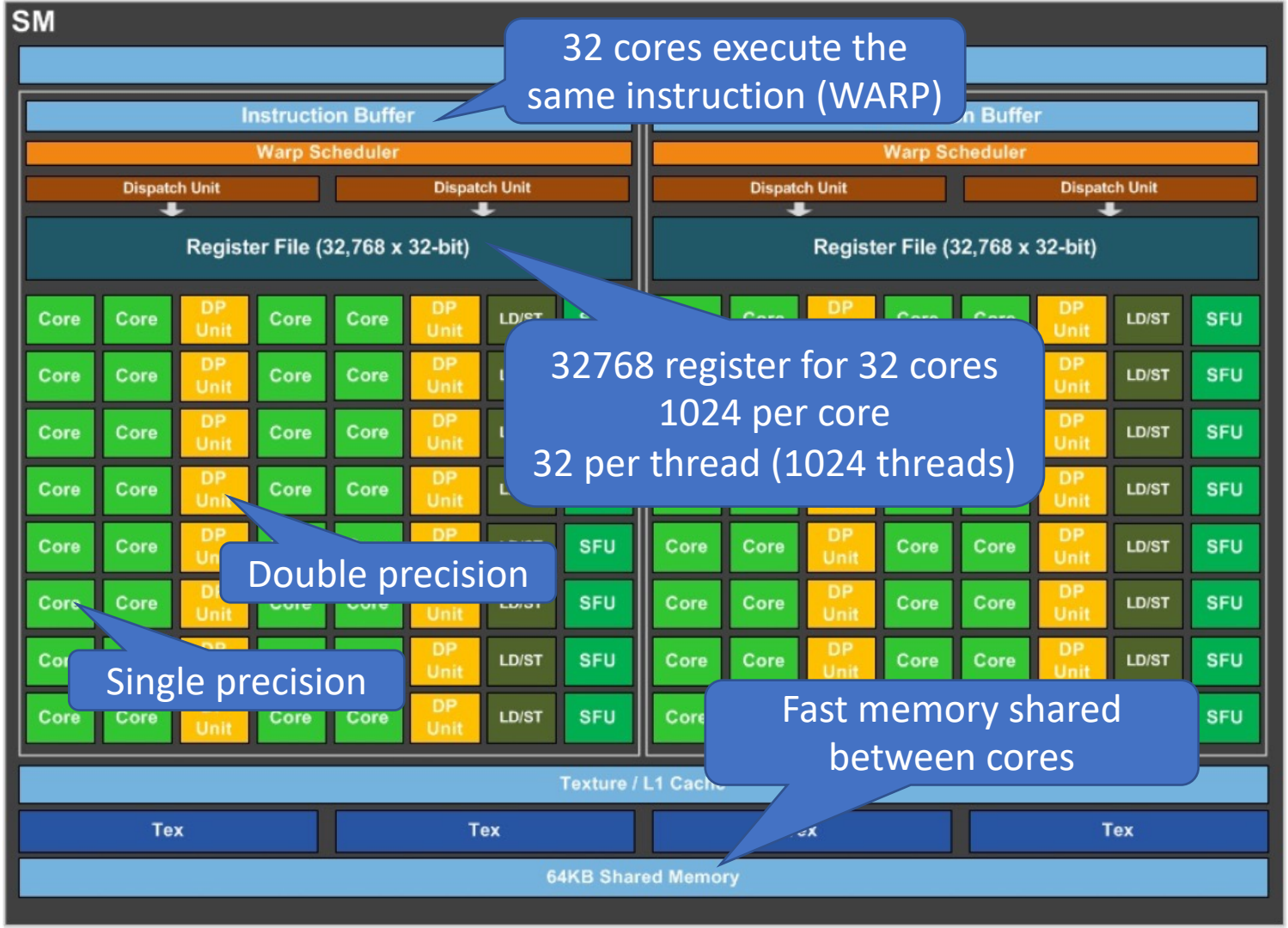

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
Streaming Multiprocessor (SM)
P100 has 56 of them
The L2 cache will reduce **latency** to global memory
L2 Cache
High-Speed Hub
NVLink

Streaming Multi-processor (SM)

P100

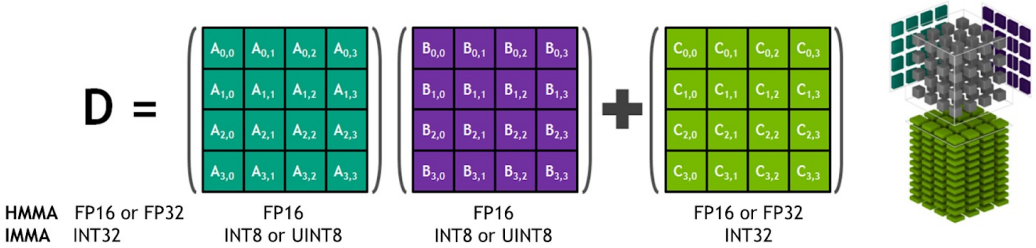
Streaming Multi-processor (SM)

P100



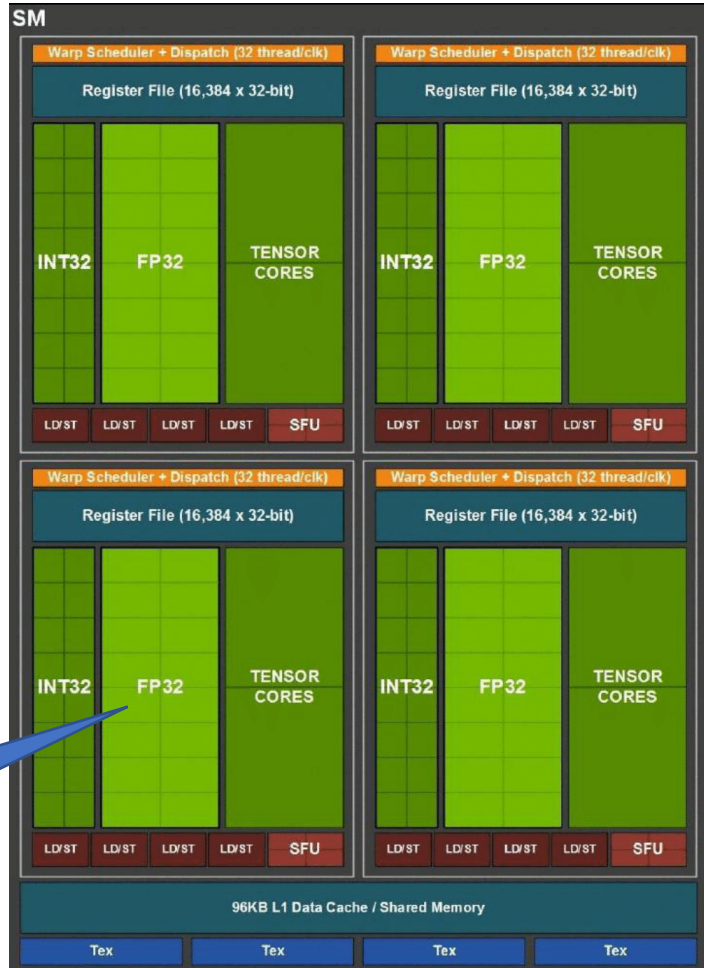
Streaming Multi-processor (SM)

A100



- 19 TF (single) versus 312 TF (tensor)
- Ideal for machine learning

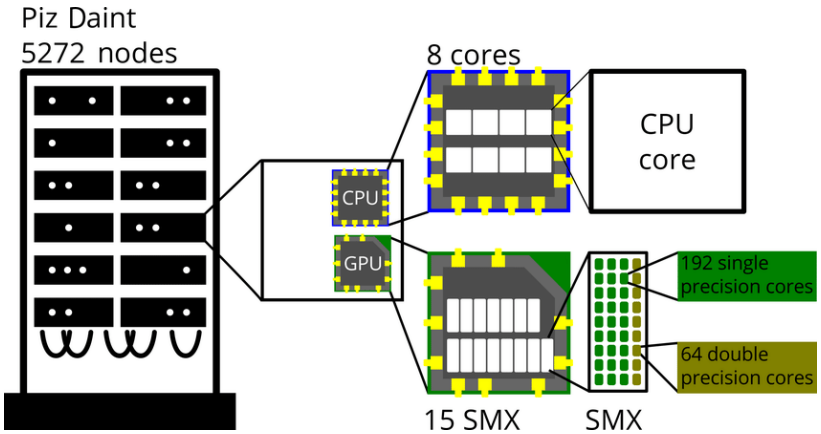
A100 vs. GeForce



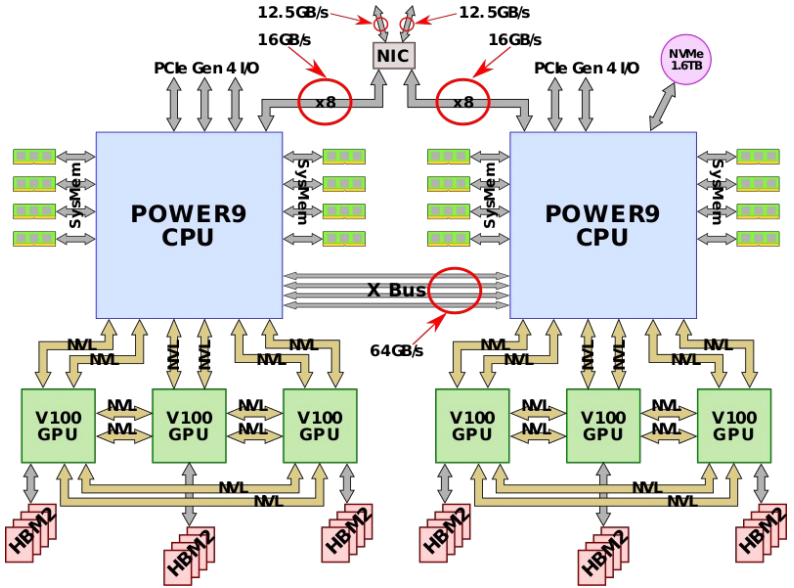
Only single precision

Supercomputer Architectures

Piz Daint

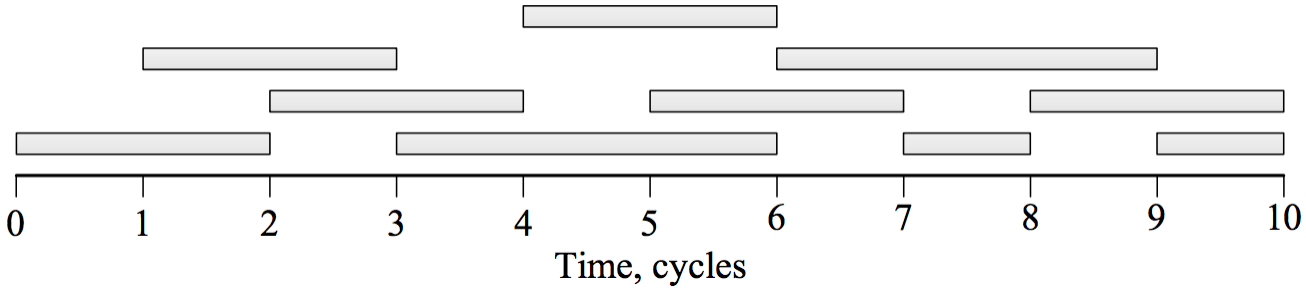


Summit



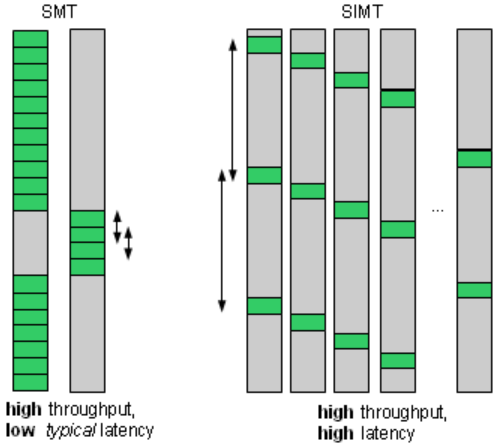
Hiding Latency

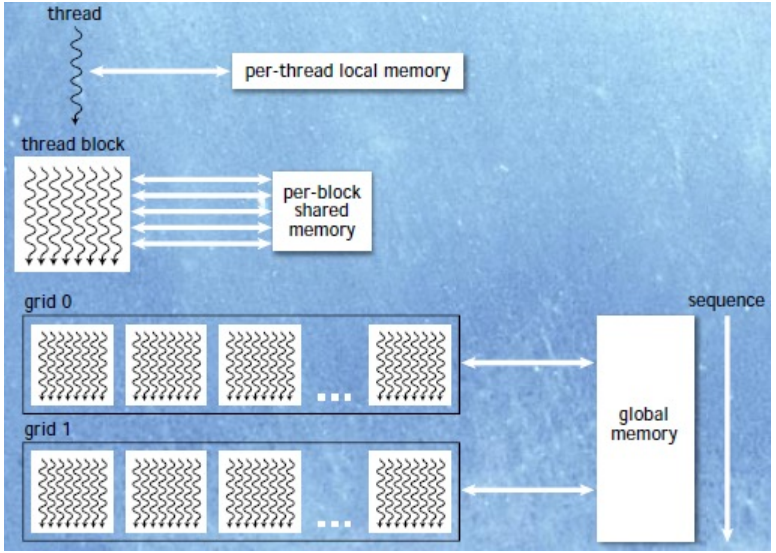
Instruction Type	Latency	Per multiprocessor	
		Peak Throughput	Needed Concurrency
Arithmetic	6 cycles	4 IPC	24 instructions
Memory Access	368 cycles	0.081 IPC	30 instructions



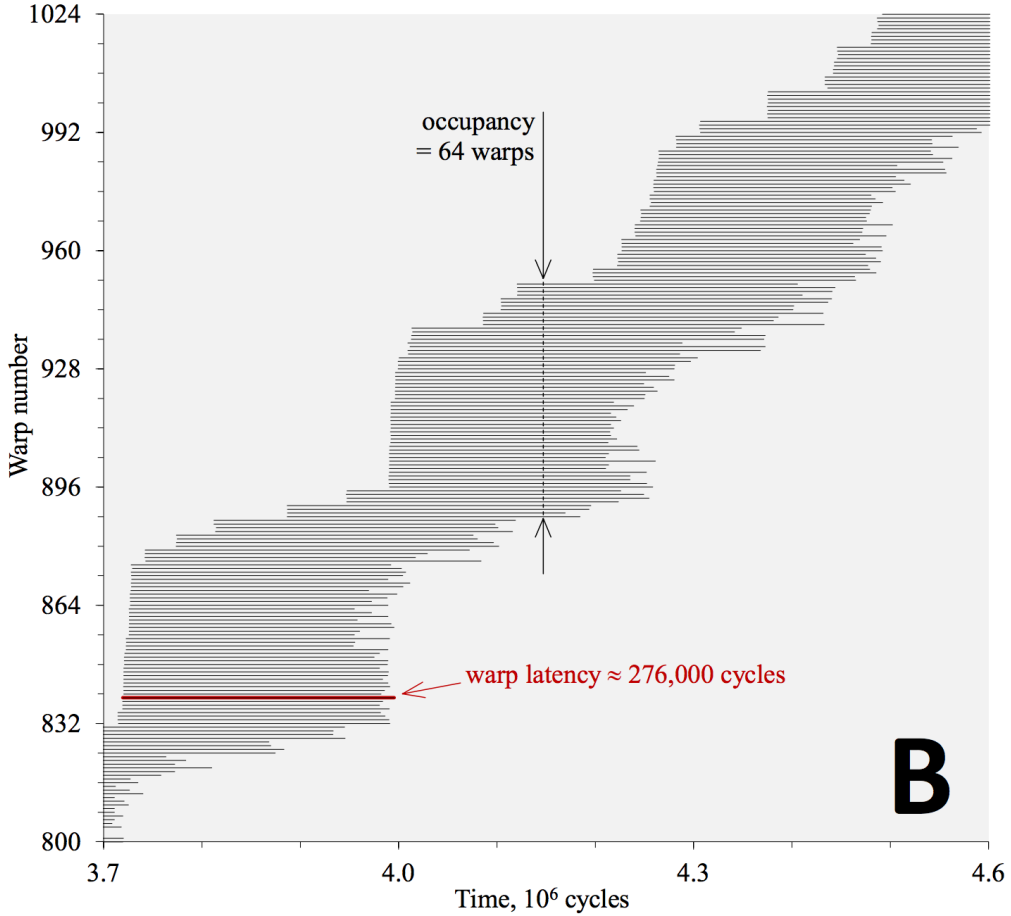
Vasily Volkov (2016)

Threads to Handle “Stalls”





Occupancy



Vasily Volkov (2016)

Achieving Peak Performance

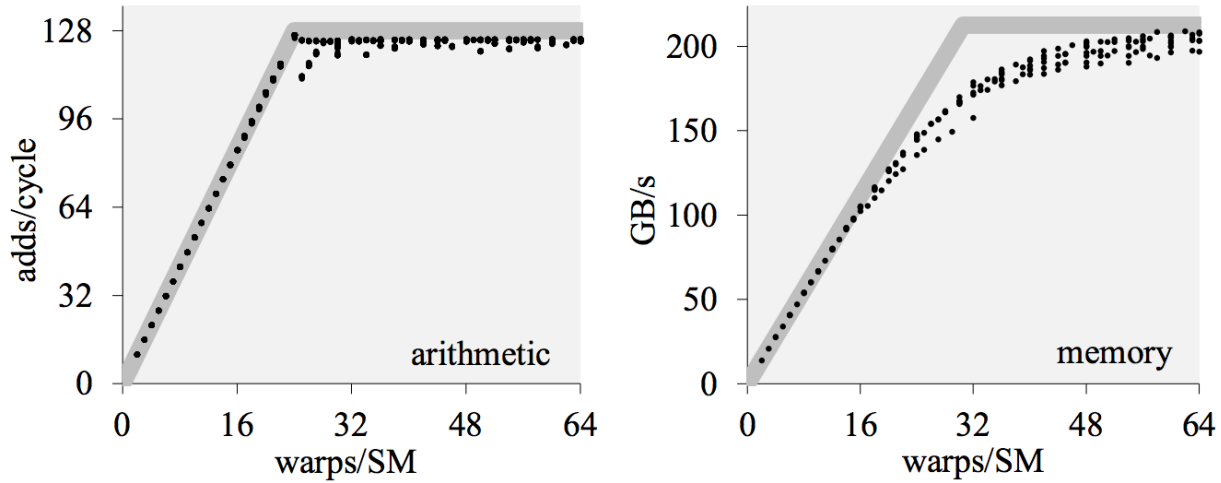


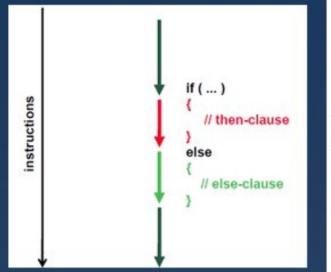
Figure 4.1: Hiding arithmetic and memory latencies on the Maxwell GPU.

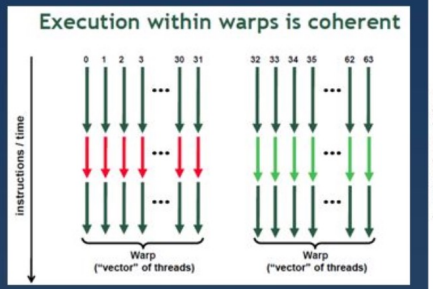
Vasily Volkov (2016)

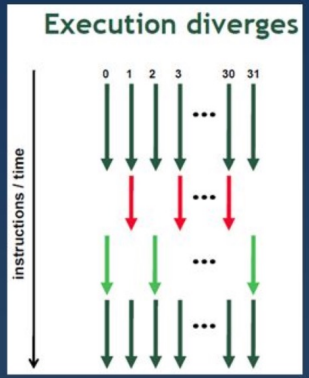
Remember Divergence?

CUDA

- Designed for 1000's of threads
  - Broken into "warps" of 32 threads
  - Entire warp runs on SM in lock step
  - Branch divergence cuts speed







Performance

12 Core AVX512 (125 Watts)

$(12 \times 16) \times 2.3 \times 2 = 883$ Gflops

3584 Core P100 (300 Watts)

$3{,}584 \times \sim 1.3 \times 2 = 9{,}300$ Gflops

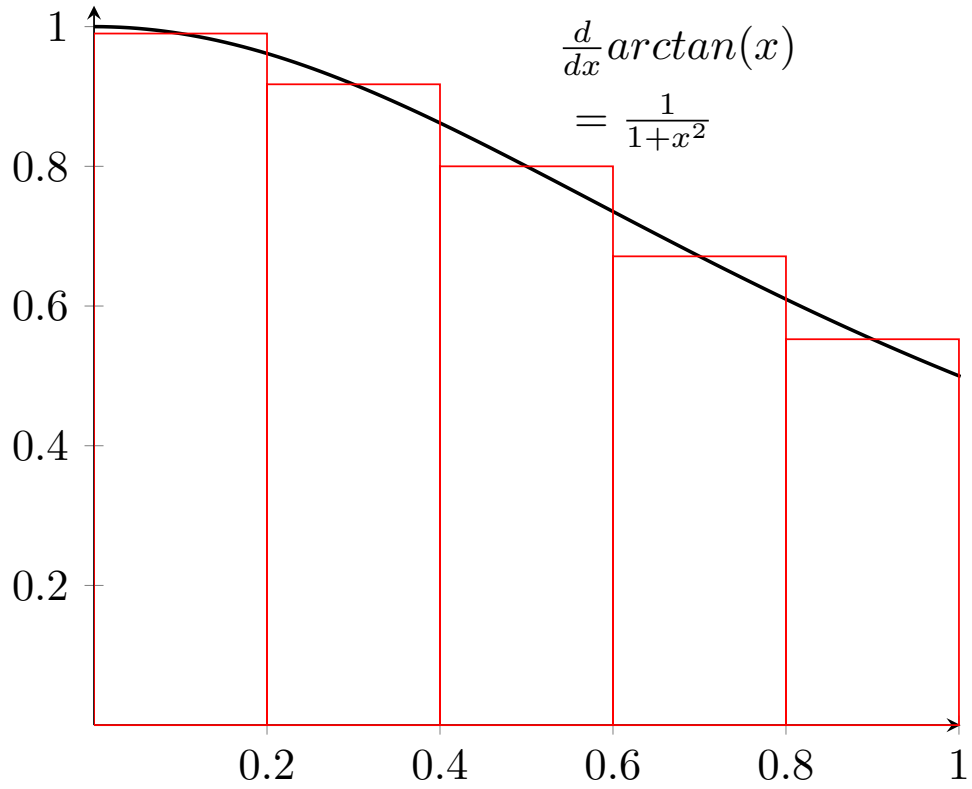
10 times faster

2.4 times as much power

Summary

- Transferring data to and from the GPU can be a bottleneck
- The “fast” global memory can be a bottleneck
- Each Warp of 32 threads does the same instruction (or nothing)
- Each SM needs many threads (up to 2048) to high latency
  - Threads share the 32,768 registers so 2048 means 16 per thread
  - Complex code can be limited by this
- Shared memory is faster and lower latency than global memory
  - Data can be shared/exchanged between threads
- BUT: writing GPU code is “easy”; performance is hard.

Integrate numerically



cpi – serial version

```
#include <stdio.h>
double getTime(void);

static long steps = 1000000000;

int main (int argc, const char *argv[]) {
    int i;
    double x;
    double pi;
    char *p;

    double step = 1.0/(double) steps;
    double sum = 0.0;
    double start = getTime();


    for (i=0; i < steps; i++) {
        x = (i+0.5)*step;
        sum += 4.0 / (1.0+x*x);
    }
    pi = step * sum;
    double delta = getTime() - start;
    printf("PI = %.16g computed in %.4g seconds\n", pi, delta);
}
```

```
srun cpi
PI = 3.141592653589768 computed in 2.347 seconds
```

cpi – OpenMP version

```
#include <stdio.h>
double getTime(void);

static long steps = 1000000000;

int main (int argc, const char *argv[]) {
    int i;
    double x;
    double pi;
    char *p;

    double step = 1.0/(double) steps;
    double sum = 0.0;
    double start = getTime();


    #pragma omp parallel for reduction(+:sum) private(x)
    for (i=0; i < steps; i++) {
        x = (i+0.5)*step;
        sum += 4.0 / (1.0+x*x);
    }
    pi = step * sum;
    double delta = getTime() - start;
    printf("PI = %.16g computed in %.4g seconds\n", pi, delta);
}
```

```
srun --export=OMP_NUM_THREADS=12 cpi_openmp
PI = 3.14159265358981 computed in 0.4518 seconds
```

cpi – OpenACC version

```
#include <stdio.h>
double getTime(void);

static long steps = 1000000000;

int main (int argc, const char *argv[]) {
    int i;
    double x;
    double pi;
    char *p;

    double step = 1.0/(double) steps;
    double sum = 0.0;
    double start = getTime();

    #pragma acc parallel
    #pragma acc loop reduction(+:sum) private(x)
    for (i=0; i < steps; i++) {
        x = (i+0.5)*step;
        sum += 4.0 / (1.0+x*x);
    }
    pi = step * sum;
    double delta = getTime() - start;
    printf("PI = %.16g computed in %.4g seconds\n", pi, delta);
}
```

```
srun cpi_openacc
PI = 3.141592653589794 computed in 0.2347 seconds
```

cpi – OpenACC repeated

```
double step = 1.0/(double) steps;
double sum = 0.0;
double start = getTime();
for(j=0;j<5;++j) {
    #pragma acc parallel
    #pragma acc loop reduction(+:sum) private(x)
    for (i=0; i < steps; i++) {
        x = (i+0.5)*step;
        sum += 4.0 / (1.0+x*x);
    }
    pi = step * sum;
    double delta = getTime() - start;
    printf("PI = %.16g computed in %.4g seconds\n", pi, delta);
}
```

```
srun cpi_openacc
PI = 3.141592653589794 computed in 0.301 seconds
PI = 3.141592653589794 computed in 0.01096 seconds
PI = 3.141592653589794 computed in 0.01085 seconds
PI = 3.141592653589794 computed in 0.01085 seconds
PI = 3.141592653589794 computed in 0.01085 seconds
```

How they were Compiled

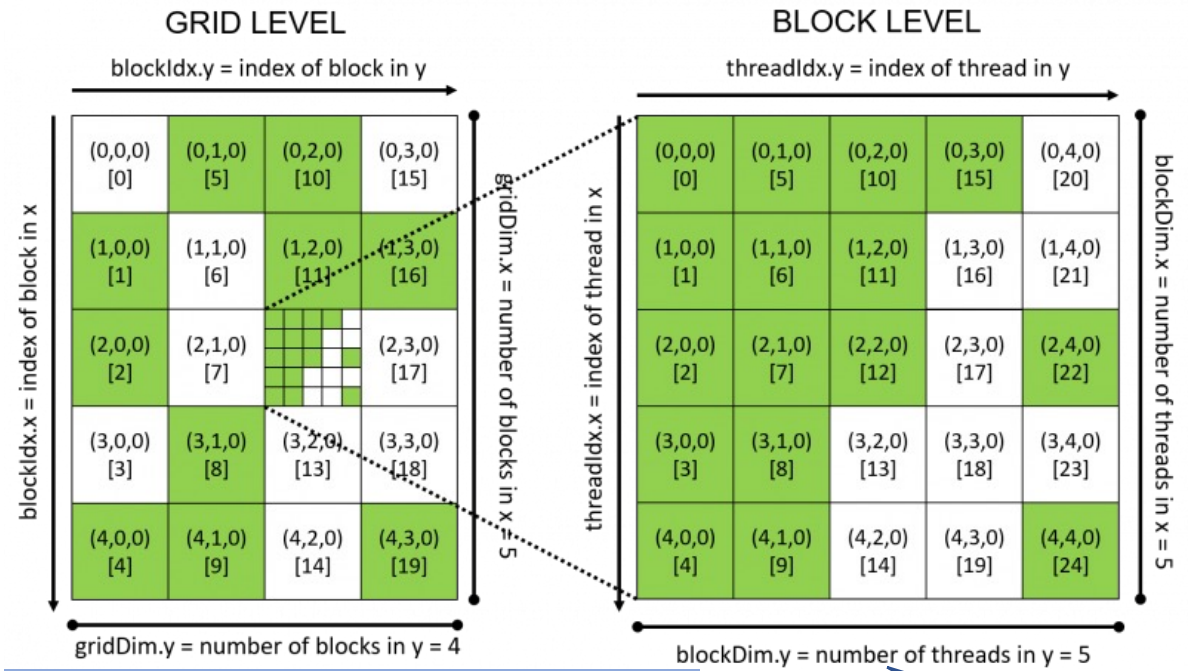
```
# Compile serial version
cc        -o cpi         cpi.c         gettime.c

# Compile OpenMP version
cc -h omp -o cpi_openmp  cpi_openmp.c  gettime.c

# Compile OpenACC version
module load daint-gpu cudatoolkit
module load craype-accel-nvidia60
cc -h acc -o cpi_openacc cpi_openacc.c gettime.c
```

cpi in CUDA

- Very simple problem
- We want to “match” the hardware (P100)
  - The P100 has “compute capability 6.0”
  - Max threads per block: 1024
  - Number of SMs: 56
- We choose how much parallelism we want
  - Number of threads per block: 64 maybe?
  - Number of blocks (number of SMs) = 500 ?

Setup

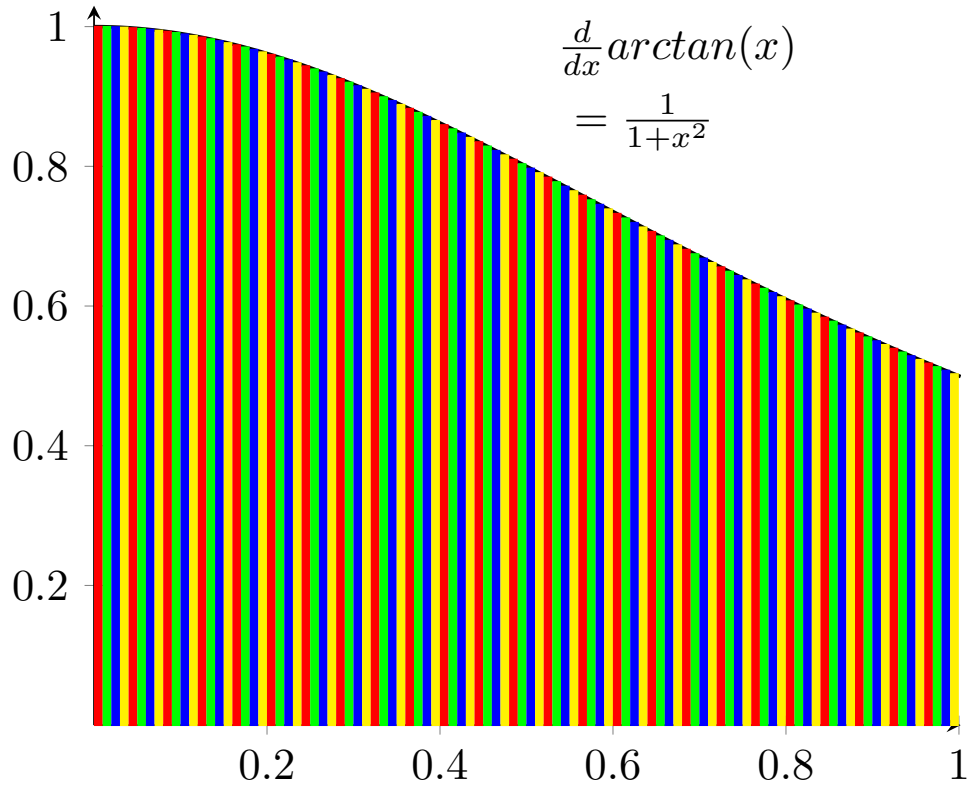
```
#define NBIN 1000000000  // Number of bins
#define NUM_BLOCK   500  // Number of thread blocks (max 2^32-1)
#define NUM_THREAD   64  // Number of threads per block (max 1024)

int main(void) {
    dim3 dimGrid(NUM_BLOCK,1,1);  // Grid dimensions
    dim3 dimBlock(NUM_THREAD,1,1);  // Block dimensions
    double *sumHost, *sumDev;  // Pointer to host & device arrays
    double pi = 0;
    int tid;

    double step = 1.0/NBIN;  // Step size
    size_t size = NUM_BLOCK*NUM_THREAD*sizeof(double);
    sumHost = (double *)malloc(size);   //  array on host
    cudaMalloc((void **) &sumDev, size);// array on GPU
    cudaMemset(sumDev, 0, size); // Zero results array
```



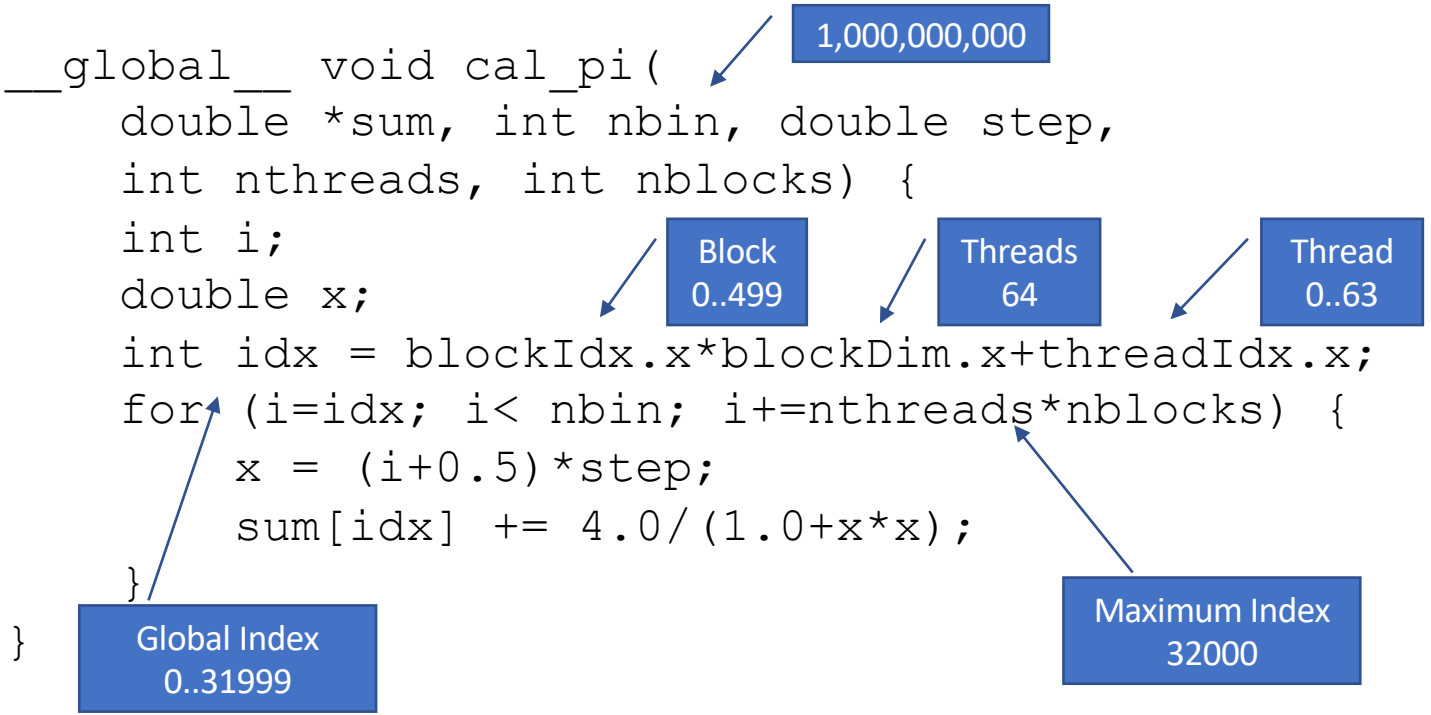

```
dim3 dimGrid(NUM_BLOCK,1,1);
dim3 dimBlock(NUM_THREAD,1,1);
```

Runs on a single SM

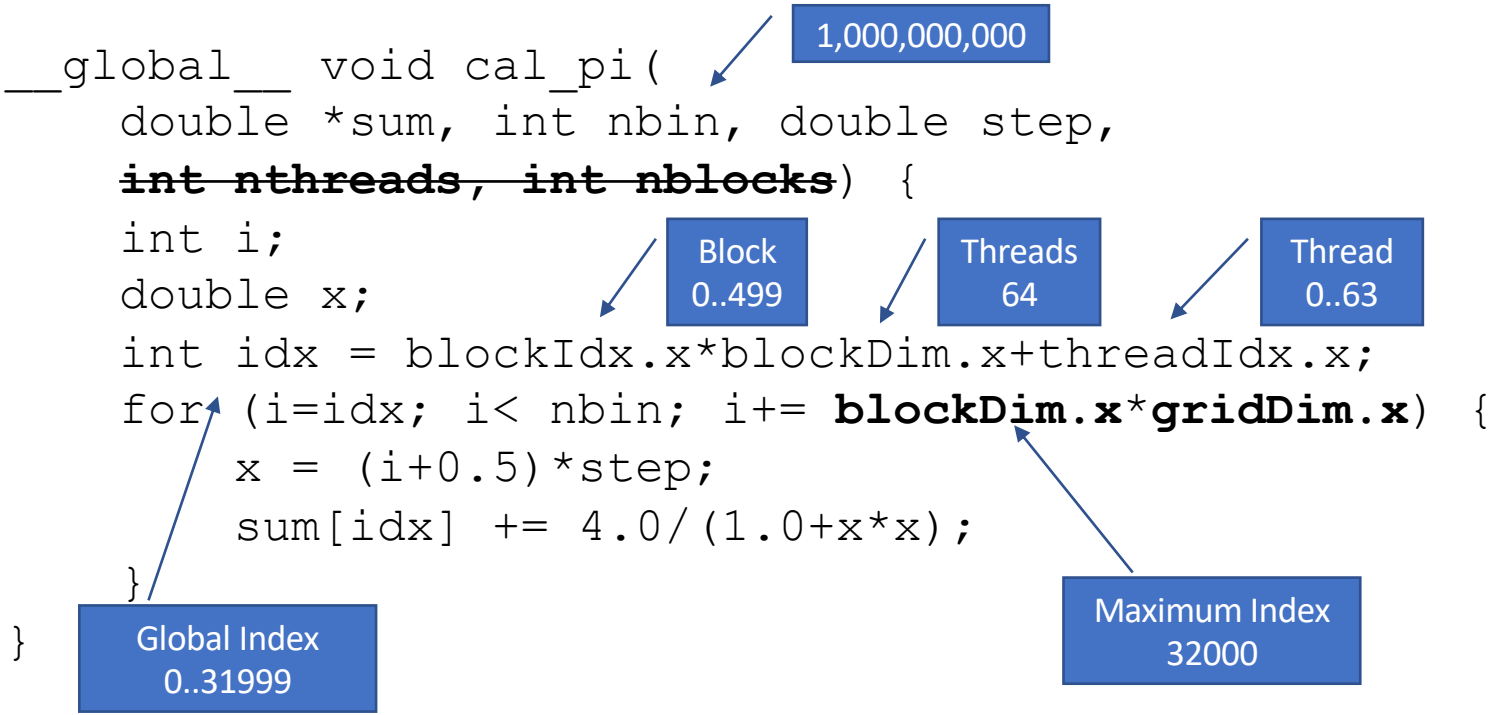
Work Split



The “kernel”


__global__ void cal_pi(
double *sum, int nbin, double step,
int nthreads, int nblocks) {
int i;
double x;
int idx = blockIdx.x*blockDim.x+threadIdx.x;
for (i=idx; i< nbin; i+=nthreads*nblocks) {
x = (i+0.5)*step;
sum[idx] += 4.0/(1.0+x*x);
}
}
1,000,000,000
Block
0..499
Threads
64
Thread
0..63
Global Index
0..31999
Maximum Index
32000

The “kernel”


1,000,000,000
__global__ void cal_pi(
double *sum, int nbin, double step,
int nthreads, int nblocks) {
int i;
double x;
int idx = blockIdx.x*blockDim.x+threadIdx.x;
for (i=idx; i< nbin; i+= blockDim.x*gridDim.x) {
x = (i+0.5)*step;
sum[idx] += 4.0/(1.0+x*x);
}
}
Block
0..499
Threads
64
Thread
0..63
Global Index
0..31999
Maximum Index
32000

Calculation

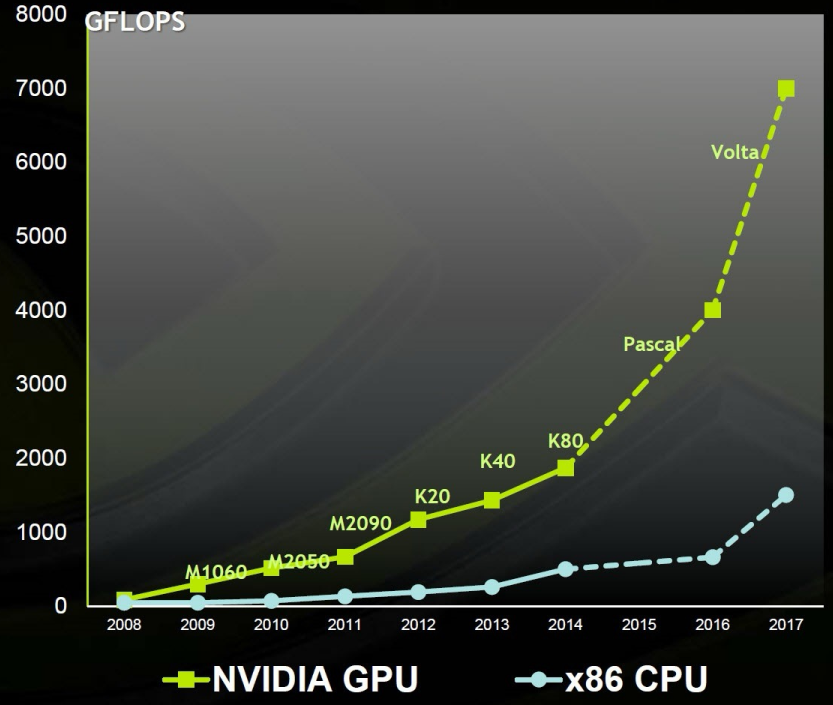
```
cal_pi <<<dimGrid, dimBlock>>>(sumDev,NBIN,step,
    NUM_THREAD, NUM_BLOCK); // call CUDA kernel
// Retrieve result from device and store it in host array
cudaMemcpy(sumHost, sumDev, size, cudaMemcpyDeviceToHost);
for(tid=0; tid<NUM_THREAD*NUM_BLOCK; tid++)
    pi += sumHost[tid];
pi *= step;

printf("PI = %.16g\n", pi);

// Cleanup
free(sumHost);
cudaFree(sumDev);
```

Results (sort by timing)

```
hpc:~> grep BLOCKS slurm-7378198.out | sort -nrk8
BLOCKS=60, THREADS=16, PI = 3.141592653589794 computed in 0.1646 seconds
BLOCKS=60, THREADS=32, PI = 3.141592656731386 computed in 0.07904 seconds
BLOCKS=120, THREADS=16, PI = 3.141592656731386 computed in 0.07412 seconds
BLOCKS=60, THREADS=48, PI = 3.141592656731386 computed in 0.05034 seconds
BLOCKS=180, THREADS=16, PI = 3.141592656731386 computed in 0.04933 seconds
...
BLOCKS=600, THREADS=160, PI = 3.141592656731378 computed in 0.008928 seconds
BLOCKS=420, THREADS=96, PI = 3.141592656731387 computed in 0.00887 seconds
BLOCKS=600, THREADS=96, PI = 3.141592656731388 computed in 0.008843 seconds
BLOCKS=540, THREADS=160, PI = 3.141592656731393 computed in 0.008833 seconds
BLOCKS=480, THREADS=128, PI = 3.141592656731393 computed in 0.008813 seconds
BLOCKS=540, THREADS=128, PI = 3.141592656731386 computed in 0.00874 seconds
BLOCKS=420, THREADS=64, PI = 3.141592656731383 computed in 0.008721 seconds
BLOCKS=600, THREADS=128, PI = 3.141592656731393 computed in 0.008679 seconds
BLOCKS=540, THREADS=96, PI = 3.141592656731391 computed in 0.008658 seconds
BLOCKS=480, THREADS=64, PI = 3.14159265673139 computed in 0.00863 seconds
BLOCKS=540, THREADS=64, PI = 3.141592656731388 computed in 0.008572 seconds
BLOCKS=600, THREADS=32, PI = 3.141592656731388 computed in 0.00853 seconds
BLOCKS=600, THREADS=64, PI = 3.14159265673139 computed in 0.008517 seconds
BLOCKS=300, THREADS=64, PI = 3.141592656731388 computed in 0.008492 seconds
```

GPU Motivation (I): Performance Trends

Peak Double Precision FLOPS



Peak Memory Bandwidth