Parallel Accelerators

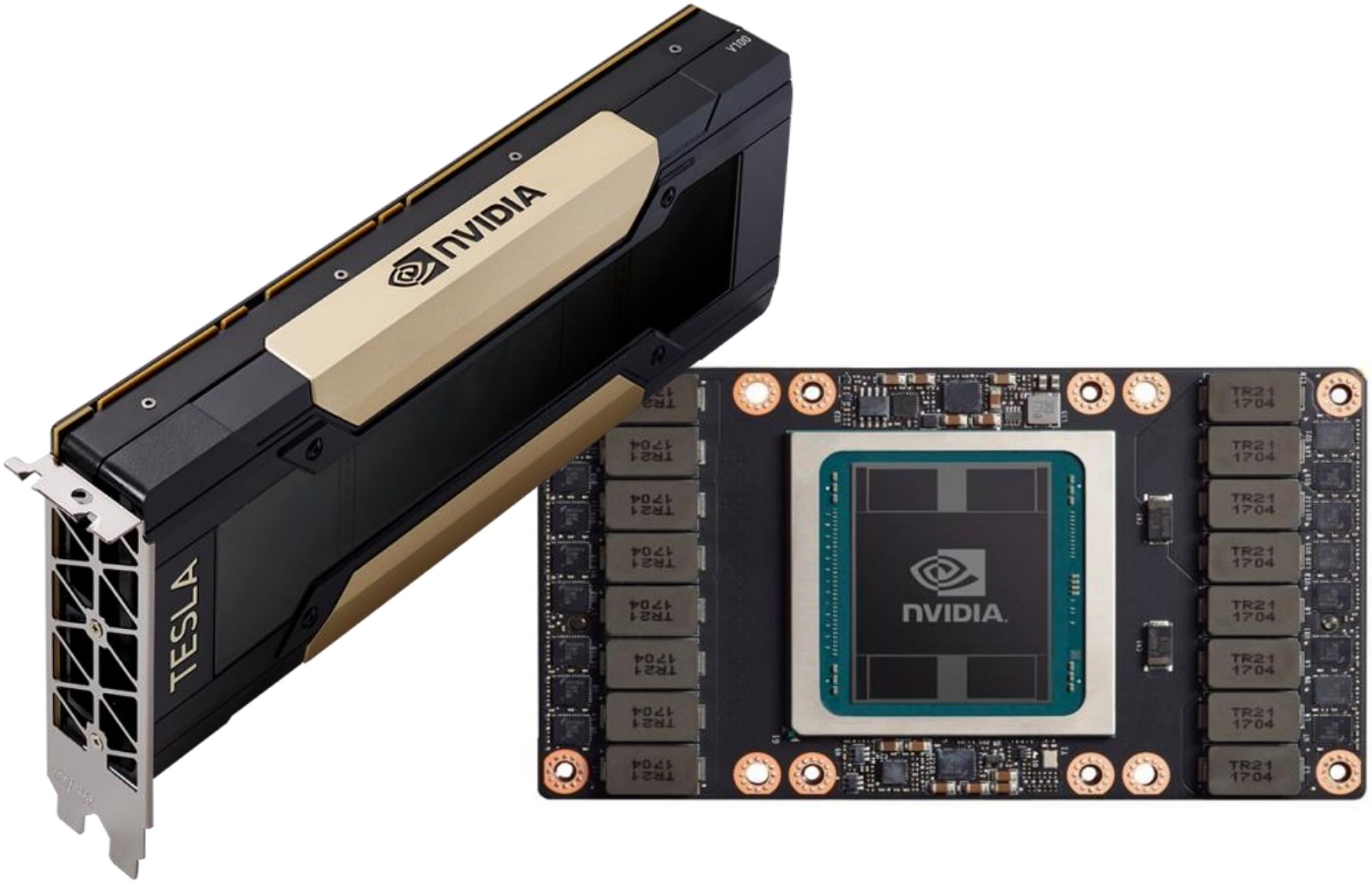
Přemysl Šůcha

``Parallel algorithms", 2017/2018
CTU/FEL

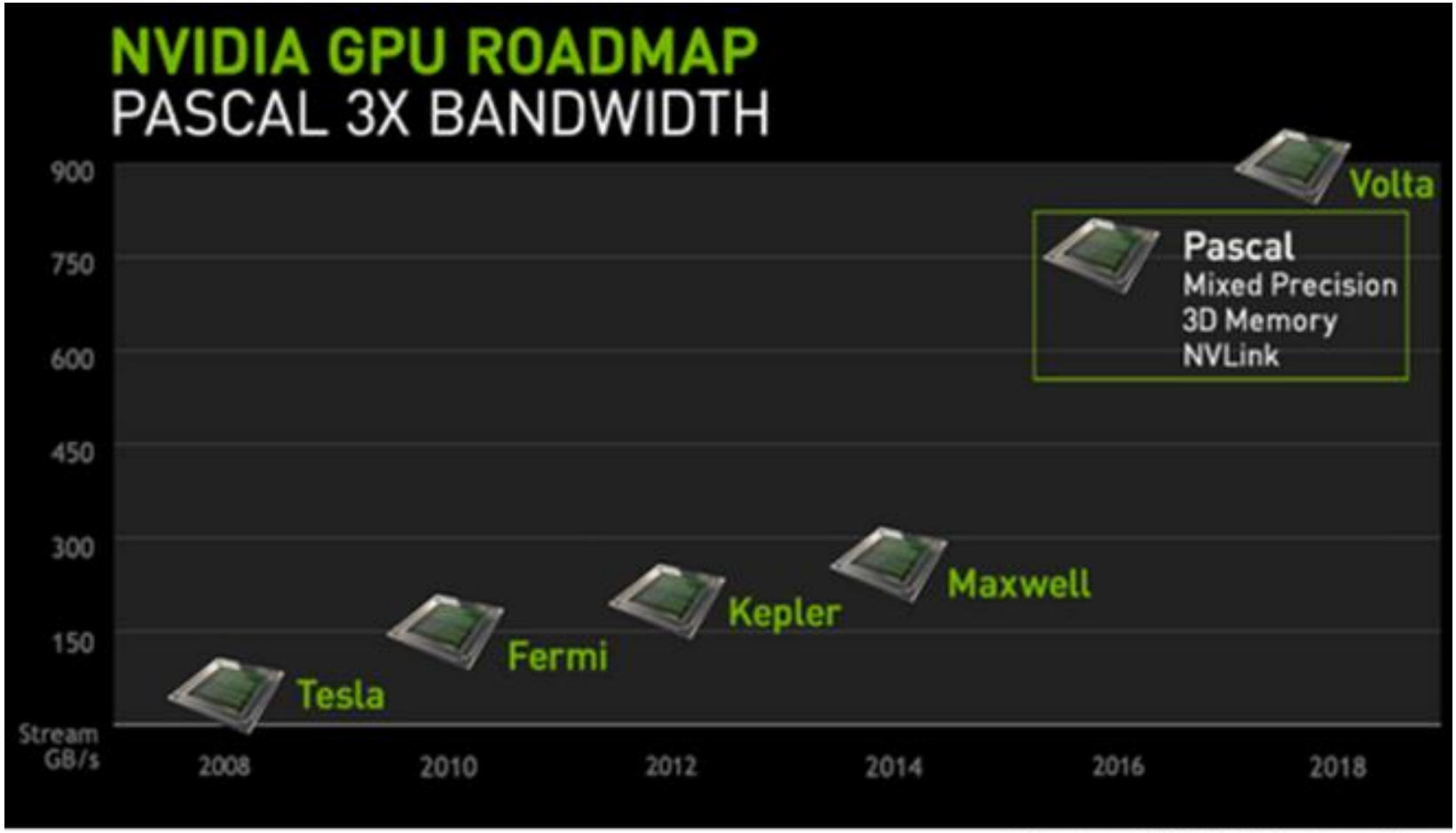
Topic Overview

- Graphical Processing Units (GPU) and CUDA
- Vector addition on CUDA
- Intel Xeon Phi
- Matrix equations on Xeon Phi

Graphical Processing Units



GPU – Nvidia - Roadmap



GPU - Use

- GPU is especially well-suited to address problems that can be expressed as **data-parallel computations**.
- The same program is executed on many data elements in parallel - with high **arithmetic intensity**.
- Applications that process **large data sets** can use a data-parallel programming model to speed up the computations (3D rendering, image processing, video encoding, …)
- Many algorithms **outside the field of image rendering and processing** are accelerated by data-parallel processing too (machine learning, general signal processing, physics simulation, finance, …).

GPU - Overview

- CPU code runs on the **host**, GPU code runs on the **device**.
- A **kernel** consists of multiple threads.
- Threads execute in 32-thread groups called **warps**.
- Threads are grouped into **blocks**.
- A collection of blocks is called a **grid**.

GPU - Hardware Organization Overview

- GPU chip consists of one or more streaming **multiprocessors** (**SM**).
- A multiprocessor consists of 1 (CC 1.x), 2 (CC 2.x), or 4 (CC 3.x, 5.x, 6.x, 7.x) **warp schedulers**. (CC = CUDA Capability)
- Each warp scheduler can issue to 2 (CC 5 and 6) or 1 (CC 7) **dispatch units**.
- A multiprocessor consists of **functional units** of several types.

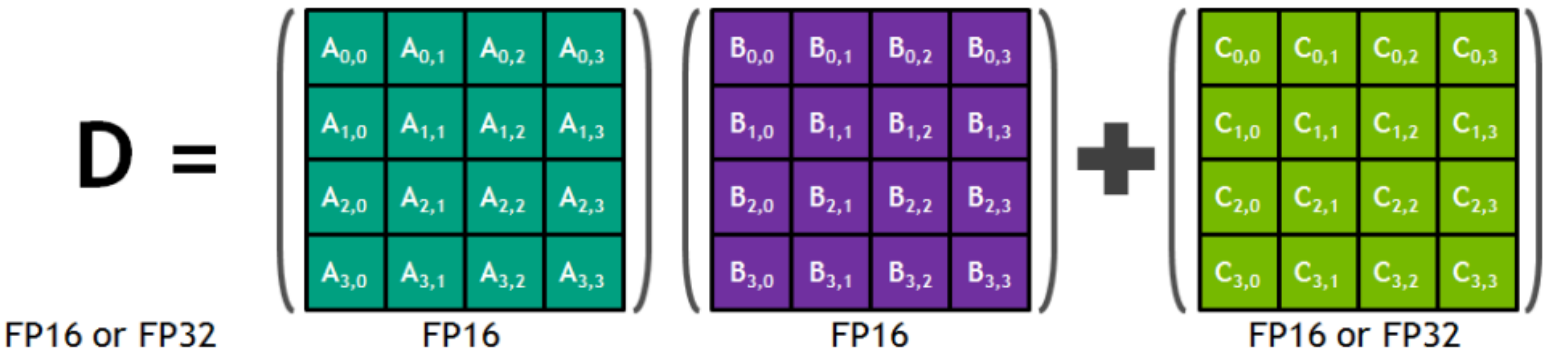
Streaming Multiprocessor (SM) - Volta

GPU - Functional Units

- **INT, FP32 (CUDA Core)** - functional units that executes most types of instructions, including most integer and single precision floating point instructions.
- **FP64 (Double Precision)** - executes double-precision floating point instructions.
- **SFU (Special Functional Unit)** - executes reciprocal and transcendental instructions such as sine, cosine, and reciprocal square root.
- **LD/ST (Load/Store Unit)** – handles load and store instructions.
- **TENSOR CORE** – for deep learning matrix arithmetic.

GPU - Tensor Core

- V100 GPU contains 640 Tensor Cores: eight (8) per SM.
- Tensor Core performs 64 floating point FMA (fused multiply–add) operations per clock.
- Matrix-Matrix multiplication (GEMM) operations are at the core of neural network training and inferencing.
- Each Tensor Core operates on a 4x4 matrices.



Streaming Multiprocessor (SM)

- Each SM has a set of temporary **registers** split amongst threads.
- Instructions can access high-speed **shared memory**.
- Instructions can access a cache-backed **constant space**.
- Instructions can access **local memory**.
- Instructions can access **global space**. (very slow in general)

GPU Architecture - Volta

GPU Architectures

Tesla Product	**Tesla K40**	**Tesla M40**	**Tesla P100**	**Tesla V100**
Manufacturing Process	28nm	28nm	16nm	12nm
Transistors	7.1 Billion	8.0 Billion	15.3 Billion	21.1 Billion
SMs / GPU	15	24	28	40
F32 Cores / SM	192	128	64	64
F32 Cores / GPU	2880	3072	3584	5120
Peak FP32 TFLOPS	5	6.8	10.6	15.7
Peak FP64 TFLOPS	1.7	0,21	5.3	7.8
GPU Boost Clock	810/870 MHz	1114 MHz	1480 MHz	1530 MHz
TDP	235 W	250 W	300 W	300 W
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Maximum TDP	244W	250W	250W	300W

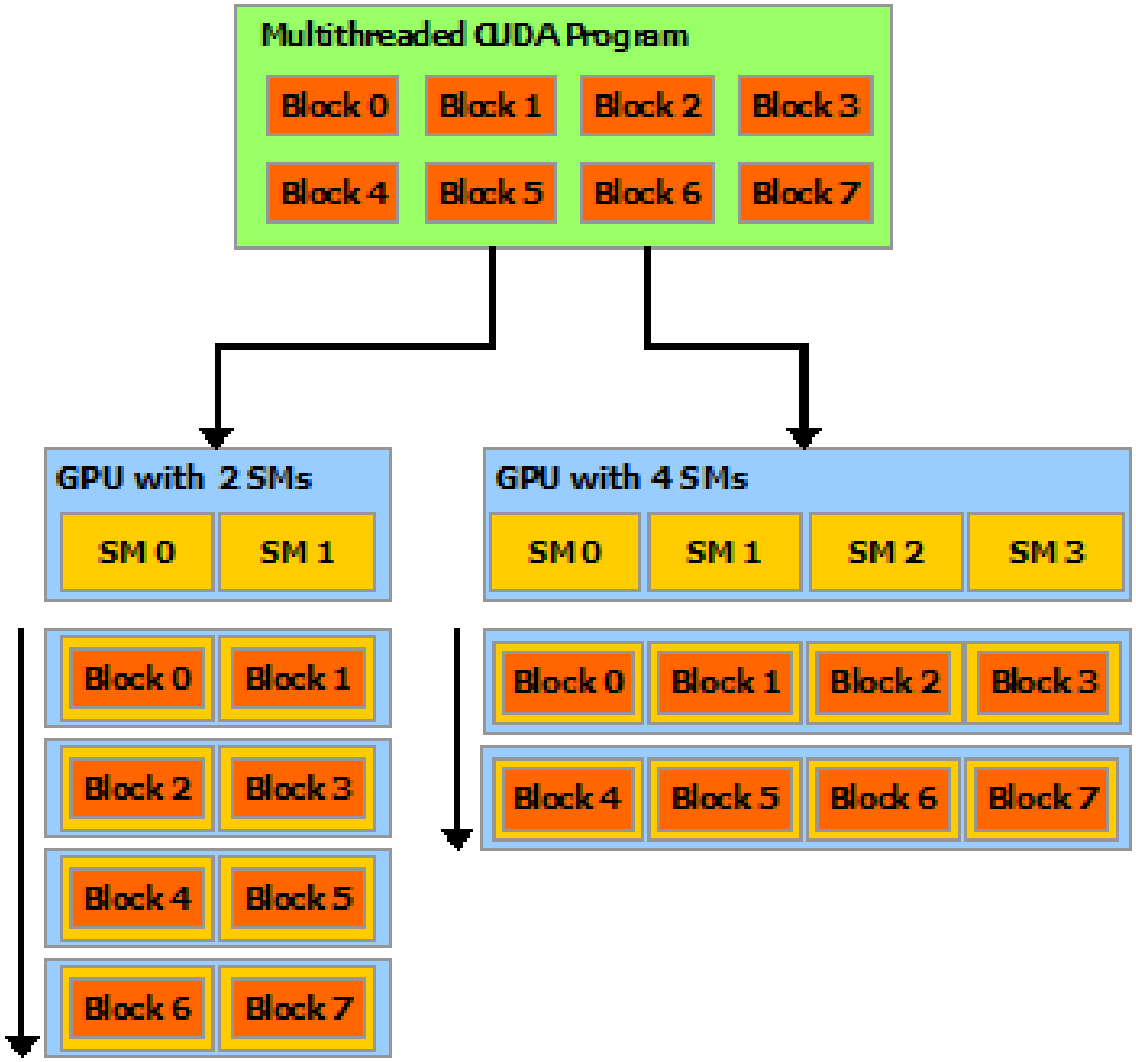
Single-Instruction, Multiple-Thread

- SIMT is an execution model where single instruction, multiple data (**SIMD**) **is combined with multithreading**.
- The SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called **warps**.
- A warp start together at the same program address, but they have their **own instruction address counter** and **register state** and are therefore **free to branch** and **execute independently**.

CUDA

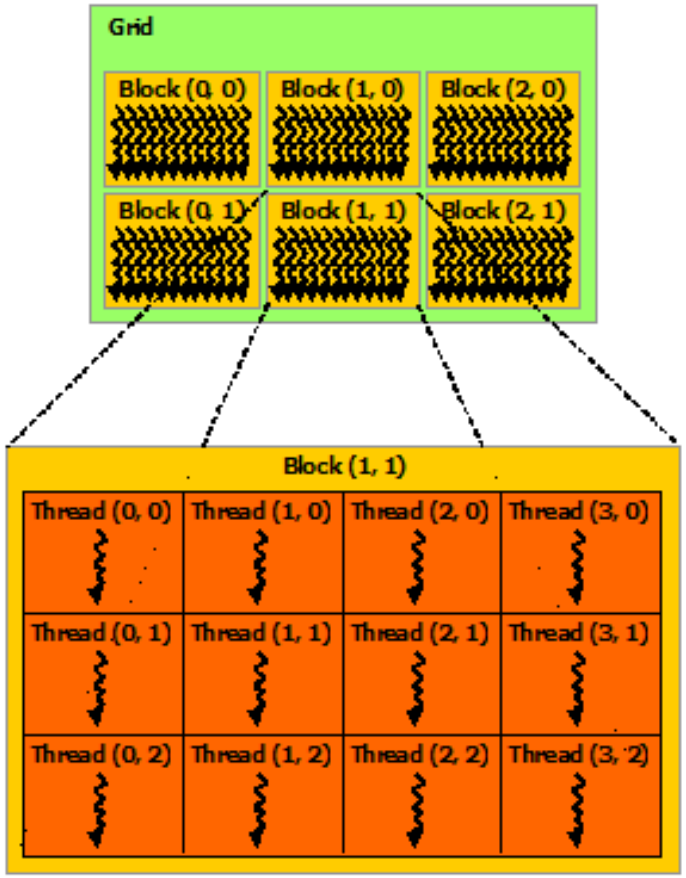
- The NVIDIA GPU architecture is built around a **scalable** array of multithreaded Streaming Multiprocessors (SMs).
- CUDA (Compute Unified Device Architecture) provides a way how a CUDA **program can be executed on any number of SMs**.
- A multithreaded program is partitioned into **blocks** of threads that execute independently from each other.
- A GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

CUDA


Multithreaded CUDA Program
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 2 SMs
SM 0
SM 1
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 4 SMs
SM 0
SM 1
SM 2
SM 3
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7

Grid/Block/Thread

- threads can be identified using a 1-D, 2-D, or 3-D thread index, forming a 1-D, 2-D, or 3-D block of threads, called a **thread block**.
- Blocks are organized into a 1-D, 2-D, or 3-D **grid** of **thread blocks**.



2-D grid with 2-D thread blocks

Kernel

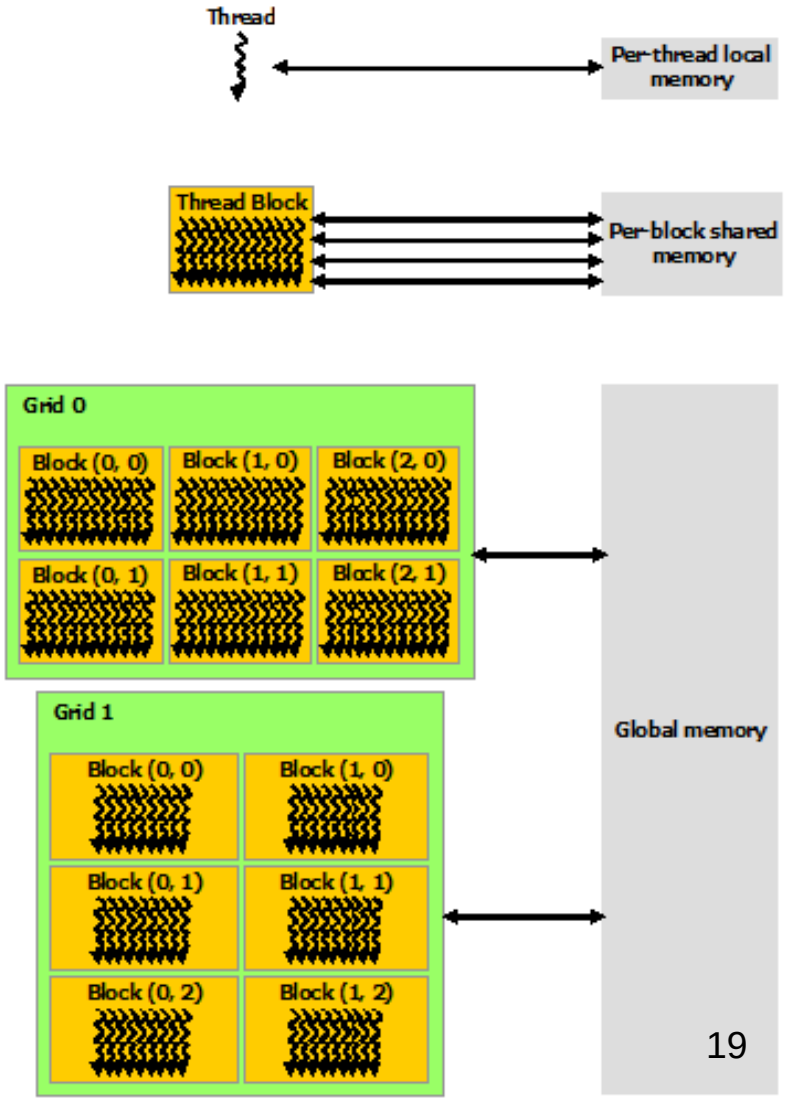
- CUDA C extends C by allowing the programmer to define C functions, called **kernels**.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}

int main()
{ ...
        // Kernel invocation with N threads inside 1 thread block
        VecAdd<<<1, N>>>(A, B, C);
}
```

- *threadIdx* is a 3-component vector, so that threads can be identified using a 1-D, 2-D, or 3-D **thread index**.

Memory Hierarchy

- Each thread has private set of **registers** and **local memory**.
- Each thread block has **shared memory** visible to all threads of the block.
- All threads have access to the same **global memory**.
- There are also two additional read-only memory spaces accessible by all threads (**constant** and **texture memory**).



GPU Programming - Example

- Element by element vector addition

[1] NVIDIA Corporation, *CUDA Toolkit Documentation v9.0.176*, 2017.

Element by element vector addition

```
/* Host main routine */
int main(void)
{
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    // Allocate the host input vectors A and B and output vector C
    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }
```

Element by element vector addition

```
// Allocate the device input vectors A and B and output vector C
float *d_A = NULL;
cudaMalloc((void **)&d_A, size);
float *d_B = NULL;
cudaMalloc((void **)&d_B, size);
float *d_C = NULL;
cudaMalloc((void **)&d_C, size);


// Copy the host input vectors A and B in host memory to the device
// input vectors in device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Element by element vector addition

```
// Launch the Vector Add CUDA Kernel
int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

// Copy the device result vector in device memory to the host result vector
// in host memory.
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

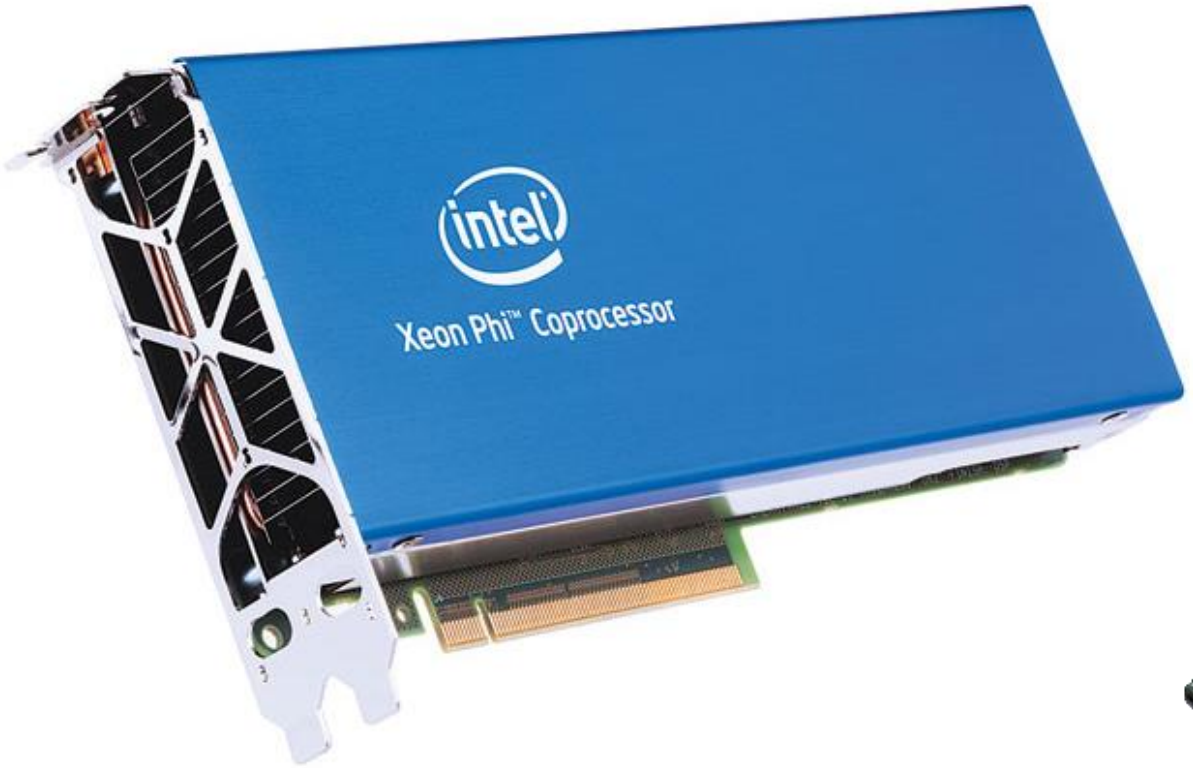
Element by element vector addition

```
    // Free device global memory
    err = cudaFree(d_A);
    err = cudaFree(d_B);
    err = cudaFree(d_C);

    // Free host memory
    free(h_A);
    free(h_B);
    free(h_C);

return 0;
}
```

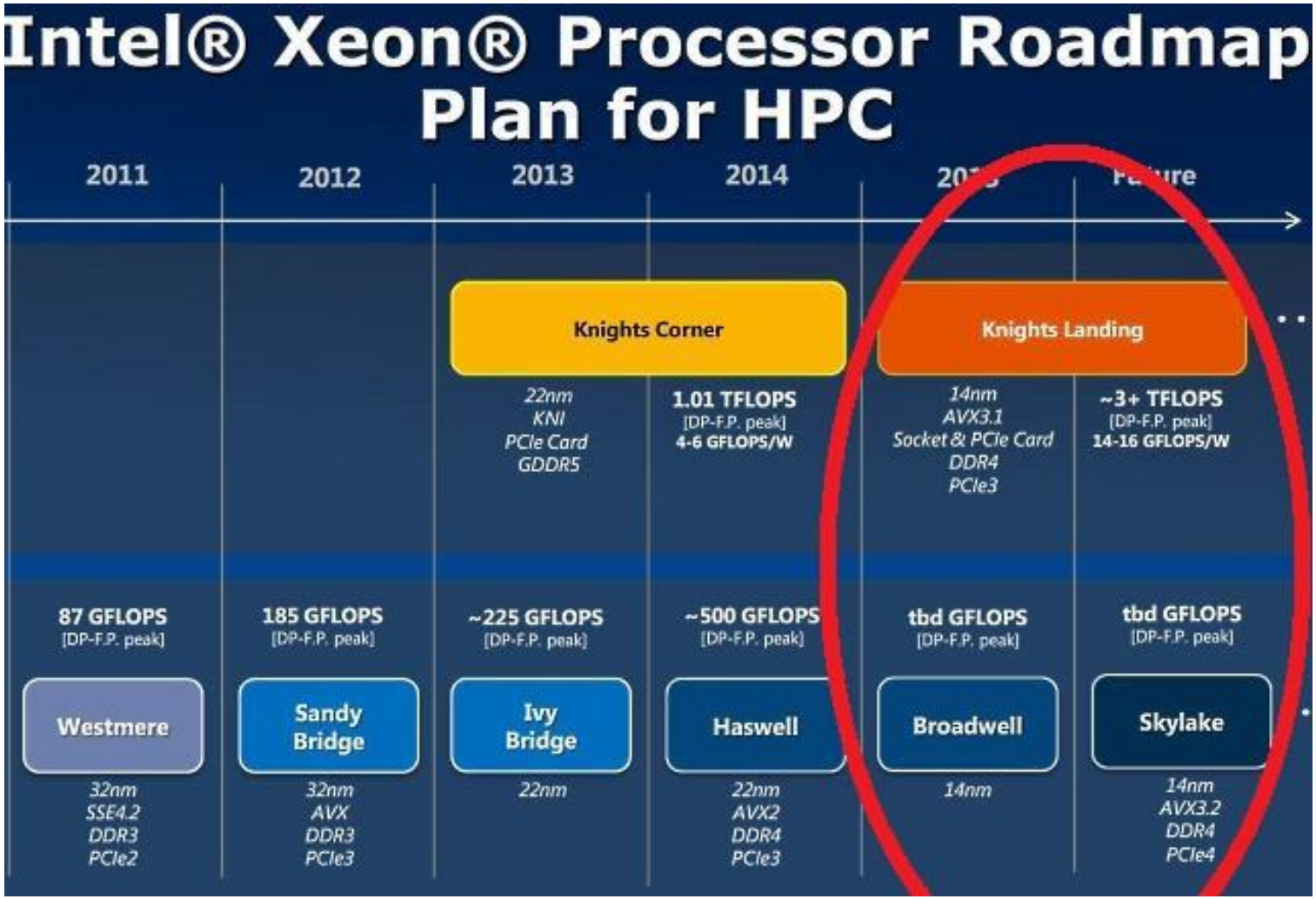
Element by element vector addition

```
/**
 * CUDA Kernel Device code
 *
 * Computes the vector addition of A and B into C. The 3 vectors have the same
 * number of elements numElements.
 */
__global__ void vectorAdd(float *A, float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```

Intel Xeon Phi



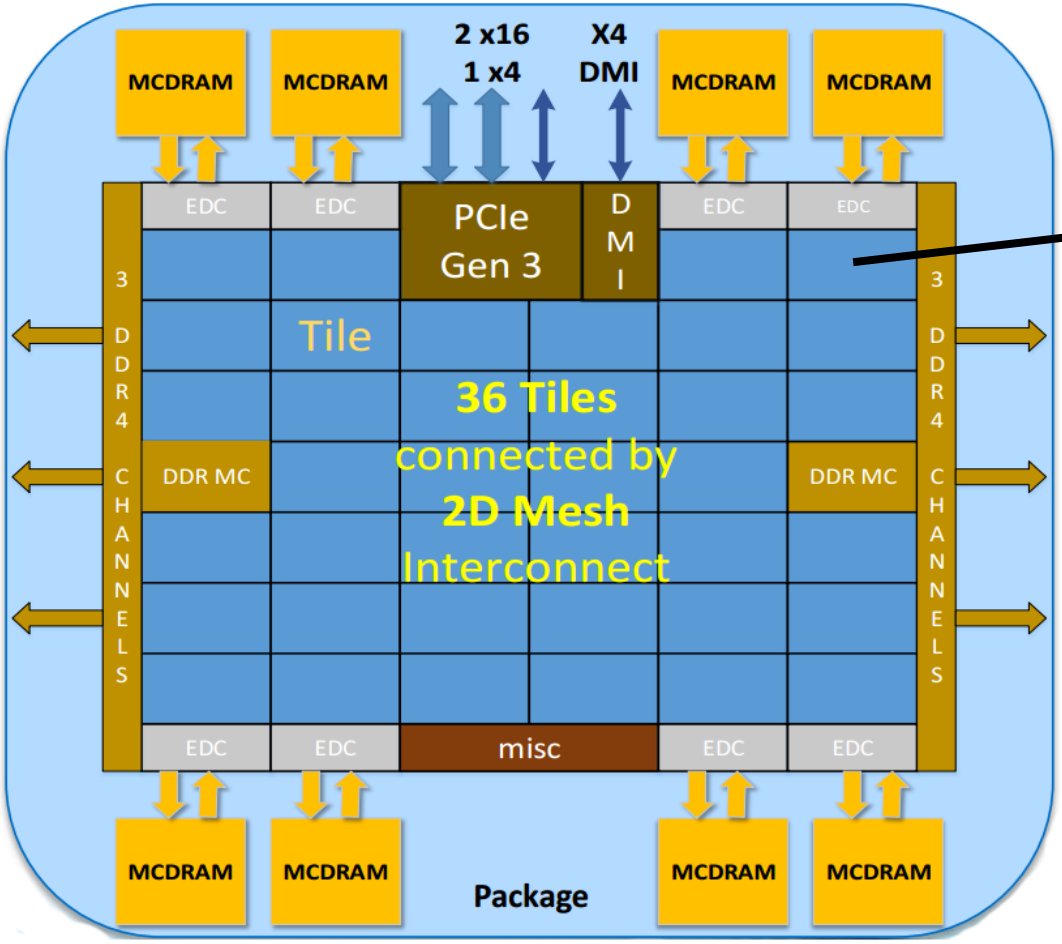
Intel Xeon Phi - Roadmap

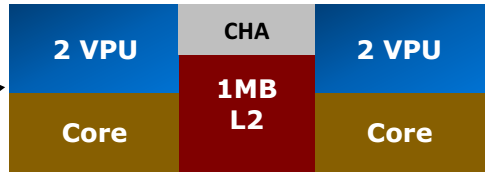


Intel Xeon Phi

- Intel Xeon Phi coprocessors are designed to **extend the reach** of applications that have demonstrated the ability to fully utilize the scaling capabilities of **Intel Xeon processor-based systems**.
- Code compiled for Xeon processors can be run on an Xeon Phi (Knights Landing).
- For successful parallelization it requires a program with **lots of threads** and operations with **vectors**.

Knights Landing Architecture





Knights Landing Architecture

- The chip is constituted by **36 tiles** interconnected by 2D mesh.
- The tile has **two Cores** (Atom Silvermont architecture), **two vector processing units** (VPU) and 1M **L2 cache**.
- A tile can execute concurrently 4 threads.
- The tiles are interconnected a **cache-coherent 2D mesh**; which provides a higher bandwidth and lower latency compare to the 1D ring interconnect on Knights corner.
- The mesh enforces **XY routing** rule.

Knights Landing Architecture

- Xeon Phi has 2 types of memory: (i) **MCDRAM** (Multi-channel DRAM) and (ii) **DDR**.
- MCDRAM is a **high-bandwidth memory** integrated on the package. There are 8 of them 2 GB each.
- MCDRAM can be configured at boot time into one of **three modes**:
  - Cache mode – MCDRAM is a cache for DDR,
  - Flat mode – MCDRAM is a standard memory in the same address space as DDR,
  - Hybrid – a combination
- DDR is a **high-capacity memory** which is external to the Knight Landing package.

Vectorization

- Each tile has **two VPUs** (Vector Processing Unit).
- It is the heard of computation. It processes all floating point computations using SSE, AVX, AVX2, …, **AVX-512**.
- Thus each tile can **execute two 512-bit vector multiple-add instructions per cycle**, i.e. compute 32 double precision resp. 64 single precision floating point operation in each cycle.

Knights Corner vs. Knights Landing

Product Name	Intel® Xeon Phi™ Coprocessor 7120X	Intel® Xeon Phi™ Processor 7290F
Code Name	Knights Corner	Knights Landing
Lithography	22 nm	14 nm
Recommended Customer Price	N/A	$6401.00
# of Cores	61	72
Processor Base Frequency	1.24 GHz	1.50 GHz
Cache	30.5 MB L2	36 MB L2
TDP	300 W	260 W
Max Memory Size (dependent on memory type)	16 GB	384 GB
Max Memory Bandwidth	352 GB/s	490 GB/s

Offloading

- Choose **highly-parallel sections** of code to run on the coprocessor. Serial code offloaded to the coprocessor will run much slower than on the CPU.
- Intel provides a set of directives to the compiler named “LEO”: **Language Extension for Offload**.
- These directives manage the transfer and execution of portions of code to the device.

```
#include <omp.h>

#pragma offload target(mic:0)
{
    //a highly-parallel section
}
```

Offloading

- A host and device don't have a shared memory, therefore **variables must be copied** in an explicit or in an implicit way.
- **Implicit copy** is assumed for scalar variables and static arrays.
- **Explicit copy** must be managed by the programmer using clauses defined in the LEO.
- Programmer clauses for explicit copy:
  in, **out**, **inout**, **nocopy**

```
#pragma offload target(mic) in(data:length(size))
```

Offloading

```
void f()
{
    int x = 55;
    int y[10] = { 0,1,2,3,4,5,6,7,8,9};
    // x, y sent from CPU
    // To use values computed into y by this offload in next offload,
    // y is brought back to the CPU
    #pragma offload target(mic:0) in(x,y) inout(y)
    { y[5] = 66; }
    // The assignment to x on the CPU
    // is independent of the value of x on the coprocessor
    x = 30;
    …
    // Reuse of x from previous offload is possible using nocopy
    // However, array y needs to be sent again from CPU
    #pragma offload target(mic:0) nocopy(x) in(y)
    {   = y[5]; // Has value 66
        = x;    // x has value 55 from first offload
    }
}
```

Xeon Phi Programming - Demo

- Simple matrix equation (*A* = *a***A* + *B*).

[2] James Jeffers and James Reinders, *Intel Xeon Phi Coprocessor High-Performance Programming*, Morgan Kaufmann, 2013.

Simple matrix equation

```
int main(int argc, char* argv[])
{

    high_resolution_clock::time_point start = high_resolution_clock::now();
    uint64_t numThreads = 1;

    #pragma offload target(mic: 0)
    {
        init();
        numThreads = omp_get_max_threads();
        #pragma omp parallel
        {
            kernel();
        }
    }
    double totalDuration = duration_cast<duration<double>>
           (high_resolution_clock::now()-start).count();
    return 0;
}
```

Simple matrix equation

```
#define IMCI_ALIGNMENT 64
#define ARR_SIZE (1024*1024)
#define ITERS_PER_LOOP 128
#define LOOP_COUNT (64*1024*1024)

#pragma omp declare target
float a;
float fa[ARR_SIZE] __attribute__((aligned(IMCI_ALIGNMENT)));
float fb[ARR_SIZE] __attribute__((aligned(IMCI_ALIGNMENT)));

inline void init()
{
    a = 1.1f;
    #pragma omp simd aligned(fa, fb: IMCI_ALIGNMENT)
    for (uint64_t i = 0; i < ARR_SIZE; ++i)
    {
        fa[i] = ((float) i) + 0.1f;
        fb[i] = ((float) i) + 0.2f;
    }
}
```

Simple matrix equation

```
inline void kernel()
{
    uint64_t offset = omp_get_thread_num()*ITERS_PER_LOOP;
    for (uint64_t j = 0; j < LOOP_COUNT; ++j)
    {
        #pragma omp simd aligned(fa,fb: IMCI_ALIGNMENT)
        for (uint64_t k = 0; k < ITERS_PER_LOOP; ++k)
            fa[k+offset] = a*fa[k+offset]+fb[k+offset];
    }
}
#pragma omp end declare target
```

References

[1] David M. Koppelman, *GPU Microarchitecture – Lecture notes*, Louisiana State University, 2017.

[2] James Jeffers and James Reinders, *Intel Xeon Phi Coprocessor High-Performance Programming*, Morgan Kaufmann, 2013.

[3] NVIDIA, *CUDA Toolkit Documentation v10.0*, 2018. (http://docs.nvidia.com/cuda/index.html)

[4] Avinash Sodani, *Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor*, Intel, 2016.

[5] James Jeffers and James Reinders and Avinash Sodani, Intel Xeon Phi Processor High Performance Programming, 2nd Edition, Morgan Kaufmann, 2016.

References

[6] Kevin, D: Effective Use of the Intel Compiler's Offload Features, https://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features, 2014.

[7] Fabio AFFINITO CINECA, Introduction to Intel Xeon Phi programming, CINECA, 2017.