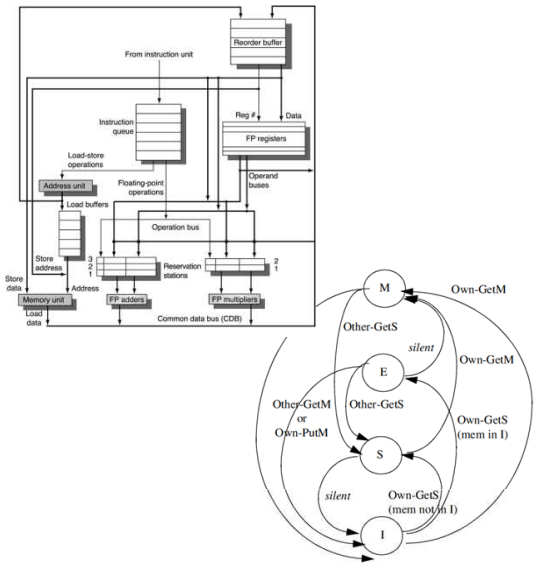
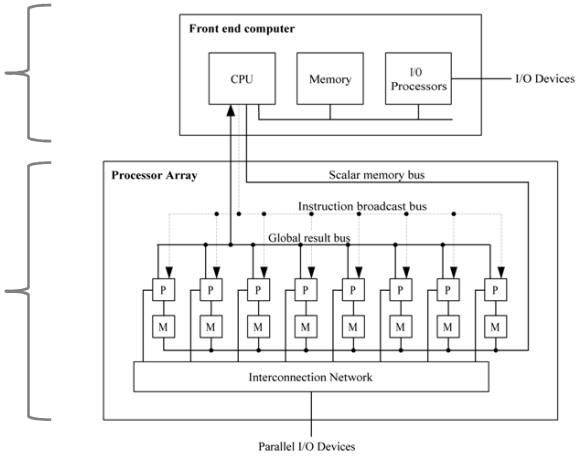
SIMD and GPU architectures

Introduction

From superscalar to SIMD and GPU architectures

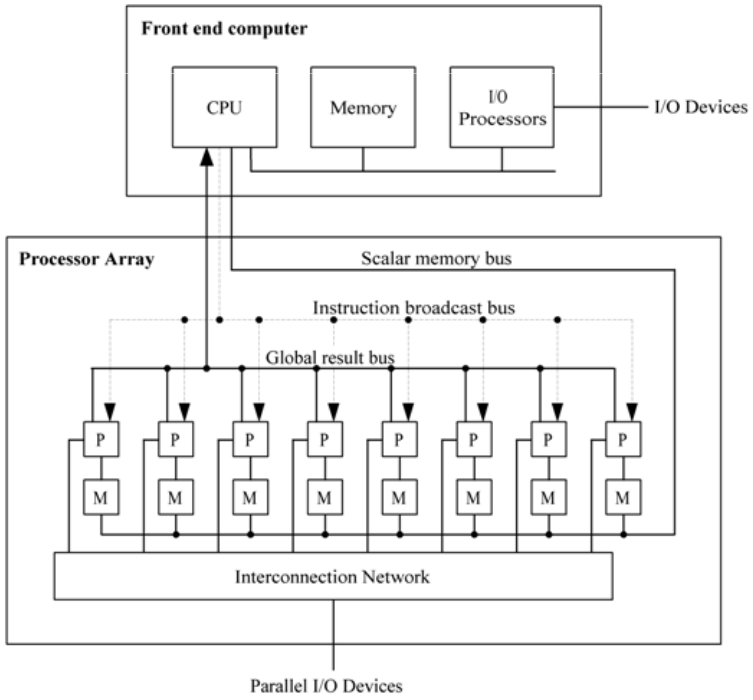
- Superscalar / dynamic scheduling / Coherence / Consistency etc…
  - single execution flow
  - very high complexity
  - too much implementation overhead
- Simultaneous MultiThreading (SMT)
  - parallelism is made visible to software
  - limited number (two/four) of *separate execution flows* sharing hardware units
- To achieve large scale parallelism:
  - rebalance *control vs. data processing*
  - most resources should go to processing
  - control should possibly be shared across execution resources



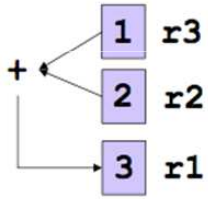


Single Instruction Multiple Data (SIMD)

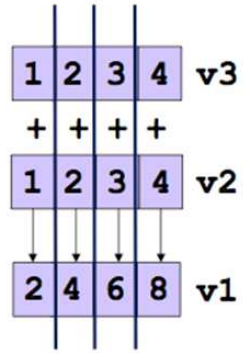
- Multiple compute units driven by the *same control*
- Processor needs to fetch and decode only one instruction
- Unlike scalar processor, SIMD operations are typically performed on vector of data



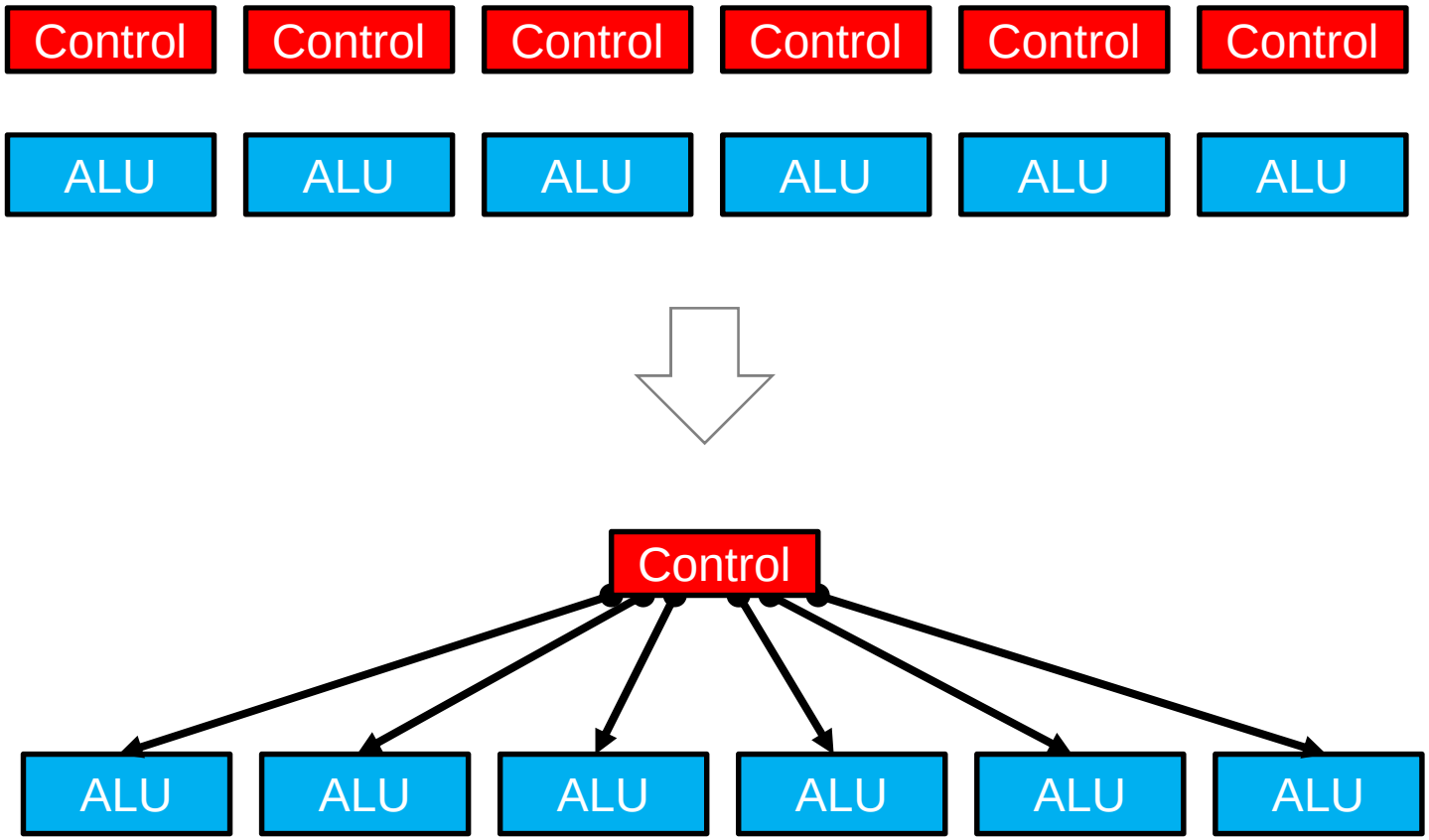

```
Scalar: add r1,r2,r3
```




```
SIMD: vadd<sws> v1,v2,v3
```

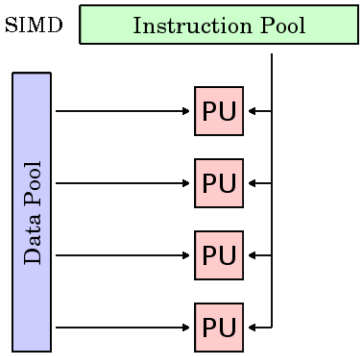


Single Instruction Multiple Data (SIMD)



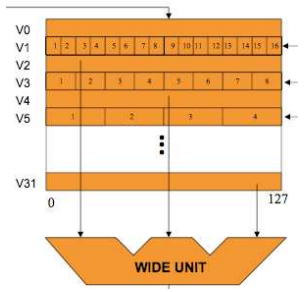
Single Instruction Multiple Data (SIMD)

- SIMD is a broad concept:



- can apply to *coarse-grain* architectures:
  - SIMD *compute systems* made of a front-end processor and an array of multiple processors working *lock-step* (i.e. doing all the same thing at the same time)

- or to *fine-grain* architectures:
  - multiple identical compute units are instantiated within the processor datapath and steered by a single control unit

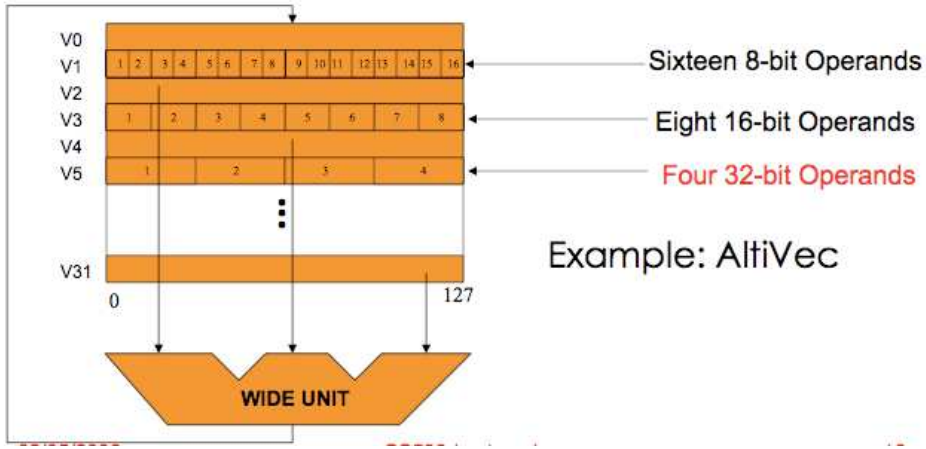


Coarse grain SIMD (processor array)

- The *front-end*, sequential processor broadcasts the commands to SIMD processors (*Processing Elements*, PEs)
  - normally, the front-end is a general-purpose CPU handling the non-parallelizable part of the program
  - When the front-end encounters a parallel task (e.g. instructions working on vectors), it issues a command to the PEs
  - although the PEs execute in parallel, some units can be allowed to skip particular instructions
- Alternately, all PEs can execute computation steps synchronously, avoiding broadcast cost to distribute results
- Each PE has a local memory not directly accessible by the control unit or other PEs
- Some PEs can be disabled, in case their portion of data must not be processed
  - All PEs work in lockstep except those that are masked out

Fine-grained SIMD (SIMD extensions)

- One processor, with multiple identical execution units
  - also called “lanes” or “slots”
- The processor has larger registers (e.g. 128 bits) that can be partitioned to store multiple data (e.g. four 32-bit values)
  - Vector length = register width / type size
  - Data fields are usually variable-sized
- The typical solution adopted by current commercial processors
  - SIMD/multimedia extensions: SIMD instructions and registers added to the base processor ISA

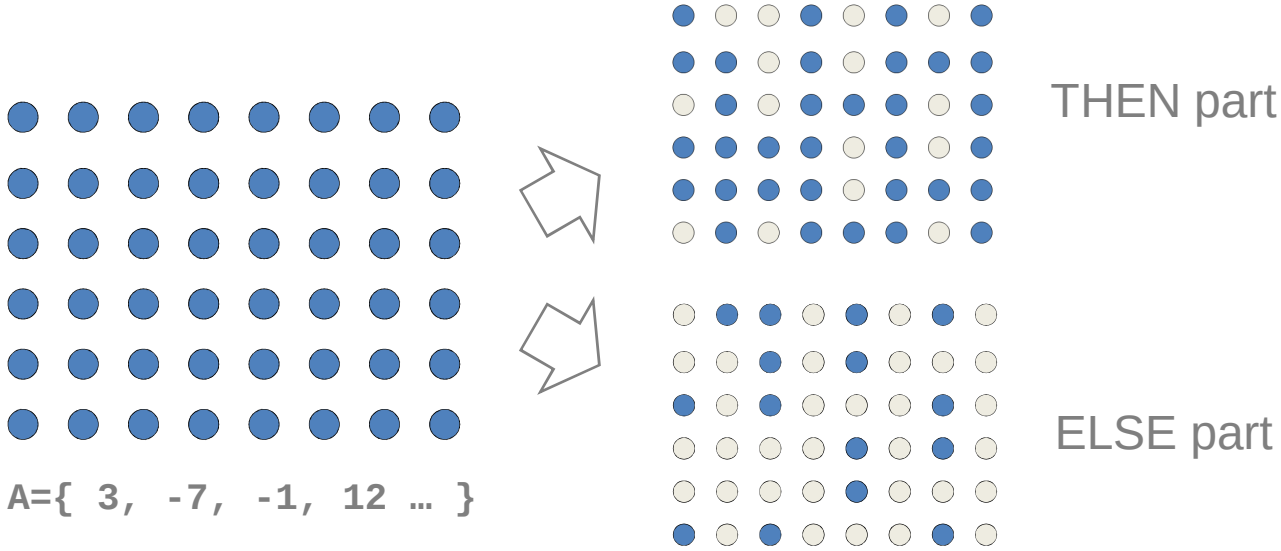


Conditional execution in SIMD architectures

- How to handle conditional *if-then-else* in processor arrays?
  - The Control Unit checks if data in each PE meets the condition
  - If it does, it sets the mask bits so those processors will participate in the operation, while the remaining ones will not
  - Unmasked processors execute the THEN part
  - Afterwards, mask bits (for original set of active processors) are flipped and unmasked processors perform the ELSE part

```
if (A>0)
        A=A-1;
else
        A=A+1;
```

A is in fact a vector, and the above operation is performed on each of its components



Advantages (and disadvantages) of SIMDs

- Alternative model for exploiting ILP
  - If code is vectorizable, then better resource and energy efficiency than out-of-order processors
    - More lanes, slower clock rate
  - Scalable if elements are independent
  - But, if there is dependency:
    - One stall per vector instruction rather than one stall per vector element
- Programmer in charge of giving hints to the compiler
- Design issues:
  - number of lanes, functional units and registers, length of vector registers, exception handling, conditional operations
- Fundamental design issue is memory bandwidth
  - Especially with virtual address translation and caching

Advantages (and disadvantages) of SIMDs

- Less hardware than multiprocessors (MIMDs):
  - only one control unit → Control units are complex
- Less memory needed than MIMD
  - Only one copy of the instructions need to be stored
  - Allows more data to be stored in memory
- Much less time required for communication between PEs and data movement
- Single instruction stream and synchronization of PEs make SIMD applications easier to program, understand, and debug
  - similar to sequential programming
- In case of a coarse-grain SIMD system, control flow and scalar operations can be executed on the front-end unit, while PEs are executing parallelized instructions
- MIMD architectures require explicit synchronization primitives
  - these may create a substantial amount of additional overhead

Advantages (and disadvantages) of SIMDs

- During a communication operation between PEs
  - PEs send data to a neighboring PE in parallel and in lock-step
  - No need to create a header with routing information since “routing” is determined by program steps
  - the entire communication operation is executed synchronously
  - SIMDs are deterministic and have much more predictable running time
  - Can normally compute a tight, worst-case upper bound to the time required for both computation and communication operations
- Less complex hardware in SIMD since no message decoder is needed in the PEs
  - MIMDs need a message decoder in each PE

- Disadvantages:
  - for applications that are not straightforward to parallelize, it may be very difficult to achieve a good utilization of the compute resources

Where are SIMD architectures used?

- Several classes of applications have data parallelism
  - Scientific and engineering applications
  - Multimedia applications
- In commercial processors, SIMD extensions are most often used for multimedia applications
  - in fact, they are typically called *multimedia extensions*, e.g. in x86
- SIMD machines typically focus on vector operations
  - Support some vector and possibly matrix operations in hardware
  - Usually limit or provide less support for non-vector type operations involving data in the “vector components”
- General purpose SIMD computers
  - May also provide some vector/matrix operations in hardware, but there is more support for traditional type (scalar) operations

Multimedia Applications and SIMD

- Short data types, narrower than the native word size
  - Graphics systems use 8 bits per primary color
  - Audio samples use 8-16 bits
  - Use a 256-bit adder for
    - 16 simultaneous operations on 16 bits
    - 32 simultaneous operations on 8 bits
- Regular data access pattern
  - Data items are contiguous in memory
- Data streaming through a series of processing stages
  - some temporal reuse for such data streams
- A few specific features that may occur sometimes:
  - many constants
  - short iteration counts
  - saturation arithmetic
  - etc.

Vector architectures

- “Vector” architectures
  - more or less the same as SIMD
  - sometimes, *vector* and *SIMD* are used interchangeably, although some authors highlight a few differences between the two
- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - send the results back to memory
- Registers are controlled by compiler
  - Register files act as compiler controlled buffers
  - Used to hide memory latency
  - Leverage memory bandwidth
- Vector loads/stores deeply pipelined
  - memory latency experienced once per vector Load/Store
  - instead, in regular architectures, memory latency is experienced once for *each element* in the vector

SIMD vs. Vector

- A few differences:
- Multimedia SIMD extensions fix the number of operands in the *opcode*
  - Vector architectures have a VLR to specify the number of operands
- Multimedia SIMD extensions
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers (in some architectures)
- The above features
  - enable vector compiler to vectorize a larger set of applications
  - make it harder for compiler to generate SIMD code and make programming in SIMD assembly more difficult
- Differences, anyway, tend to be subtle
  - SIMD extensions are increasingly being extended by manufacturers (e.g., see Intel's MMX → SSE → AVX)

SIMD vs. Vector

- SIMD meant for direct use by programmers
  - rather than for automated code generation by compilers
  - although there are “SIMD-izing” compilers, i.e. compilers extracting parallel SIMD patterns from sequential code
- Recent x86 compilers
  - Capable for FP intensive apps
- Why is SIMD popular?
  - Little hardware costs and complexity
  - Need smaller memory bandwidth than vector
  - Separate data transfers aligned in memory
    - with Vector: a single instruction might cause 64 memory accesses, making it very likely to incur a page fault in the middle of the vector!
  - Use much smaller register space
  - Fewer operands
  - No need for sophisticated mechanisms of vector architecture

SIMD example (MIPS)

- Example (DXPY):

```
L.D       F0,a          ;load scalar a
MOV       F1, F0        ;copy a into F1 for SIMD MUL
MOV       F2, F0        ;copy a into F2 for SIMD MUL
MOV       F3, F0        ;copy a into F3 for SIMD MUL
DADDIU    R4,Rx,#512    ;last address to load
Loop:
L.4D      F4,0[Rx]      ;load X[i], X[i+1], X[i+2], X[i+3]
MUL.4D    F4,F4,F0      ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
L.4D      F8,0[Ry]      ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
ADD.4D    F8,F8,F4      ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
S.4D      0[Ry],F8      ;store into Y[i], Y[i+1],Y[i+2],Y[i+3]
DADDIU    Rx,Rx,#32     ;increment index to X
DADDIU    Ry,Ry,#32     ;increment index to Y
DSUBU     R20,R4,Rx     ;compute bound
BNEZ      R20,Loop      ;check if done
```

Commercial SIMD architectures

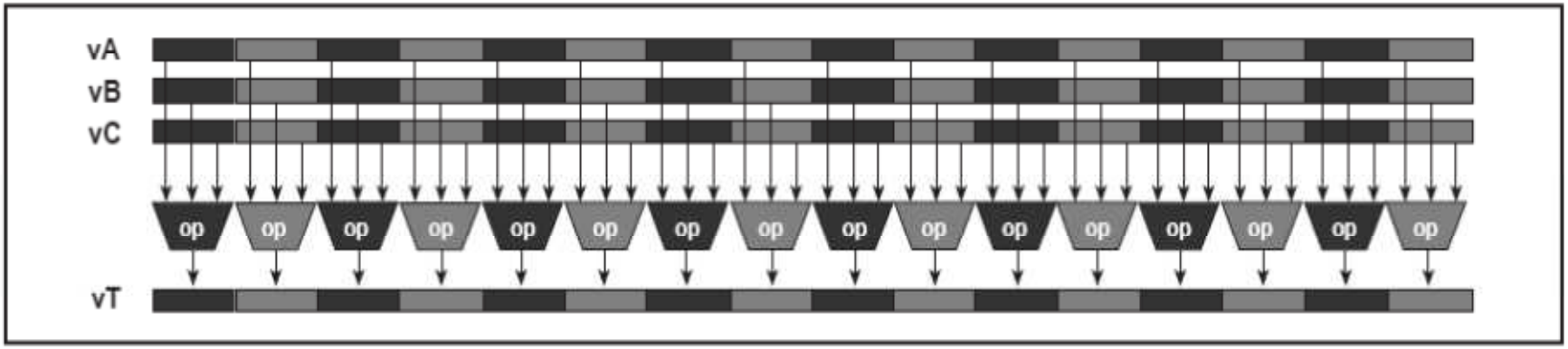
- Intel SIMD extensions
  - MMX: Multimedia Extensions (1996)
  - SSE: Streaming SIMD Extensions (1999)
  - AVX: Advanced Vector Extension (2010)
- PowerPC
  - Altivec
- ARM
  - NEON
- MIPS
  - MIPS-3D

Motorola/Freescale ALTIVEC

- Altivec is the Motorola implementation of SIMD
  - vector unit handles multiple pieces of data simultaneously in parallel with a single instruction
- added 162 new PowerPC instructions
  - functionality similar to what is offered in the scalar units, just extrapolated into the SIMD domain
  - new instructions for field permutation and formatting
  - load/store instruction options for cache management
  - instructions that control four data prefetch engines
- AltiVec vector unit never generates exceptions

Motorola/Freescale ALTIVEC

- “vector” instructions:
  - each AltiVec instruction specifies up to three source operands and a single destination operand
- Target applications for AltiVec technology include:
  - image and video processing systems
  - virtual reality
  - scientific array processing systems
  - network infrastructure such as Internet routers, etc...



Motorola/Freescale ALTIVEC

- Four 128-bit vector execution units:
  - VIU1: executes AltiVec simple integer instructions
  - VIU2: executes AltiVec complex integer instructions
  - VPU: executes AltiVec permute instructions
  - VFPU: executes AltiVec floating-point instructions
- 32-entry, 128-bit vector register file (VRs)
- 16-entry, 128-bit renamed buffer

Unit	Latency
VIU1	1
VIU2	4
VFPU	4
VPU	2

Motorola/Freescale ALTIVEC

- *Vector registers* (VRs)
  - used as source and destination operands for AltiVec load, store, and computational instructions
- AltiVec's 128-bit wide vectors can be subdivided into:
  - 16 elements, where each element is either an 8-bit signed or unsigned integer, or an 8-bit character
  - 8 elements, where each element is a 16-bit signed or unsigned integer
  - 4 elements, where each element is either a 32-bit signed or unsigned integer, or a single precision (32-bit) IEEE floating-point number

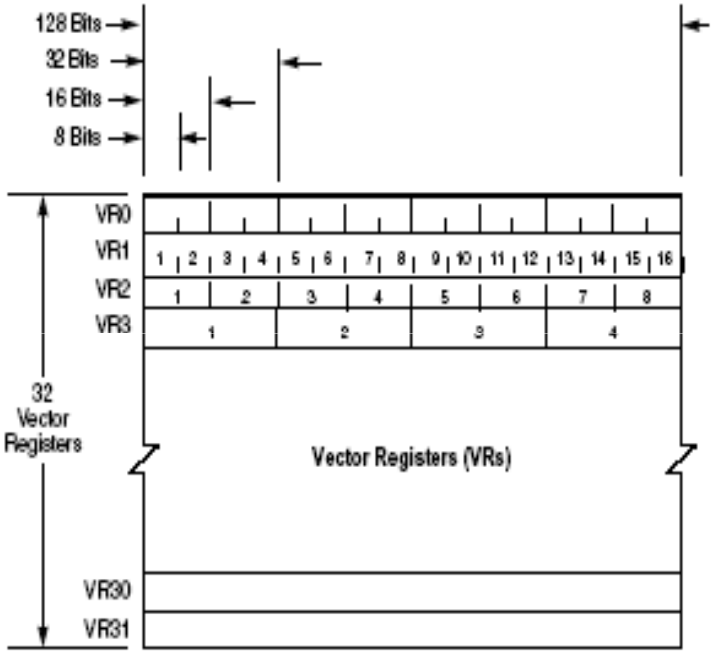
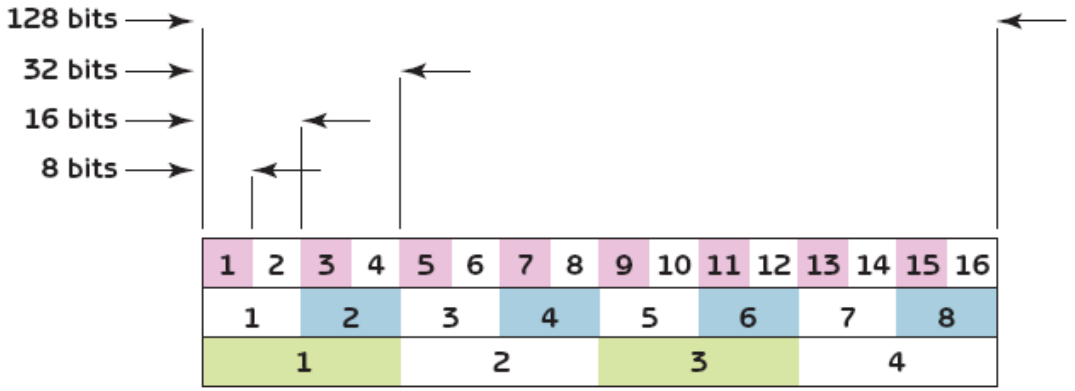


Figure 7-1. Vector Registers (VRs)

Motorola/Freescale ALTIVEC

- Data types

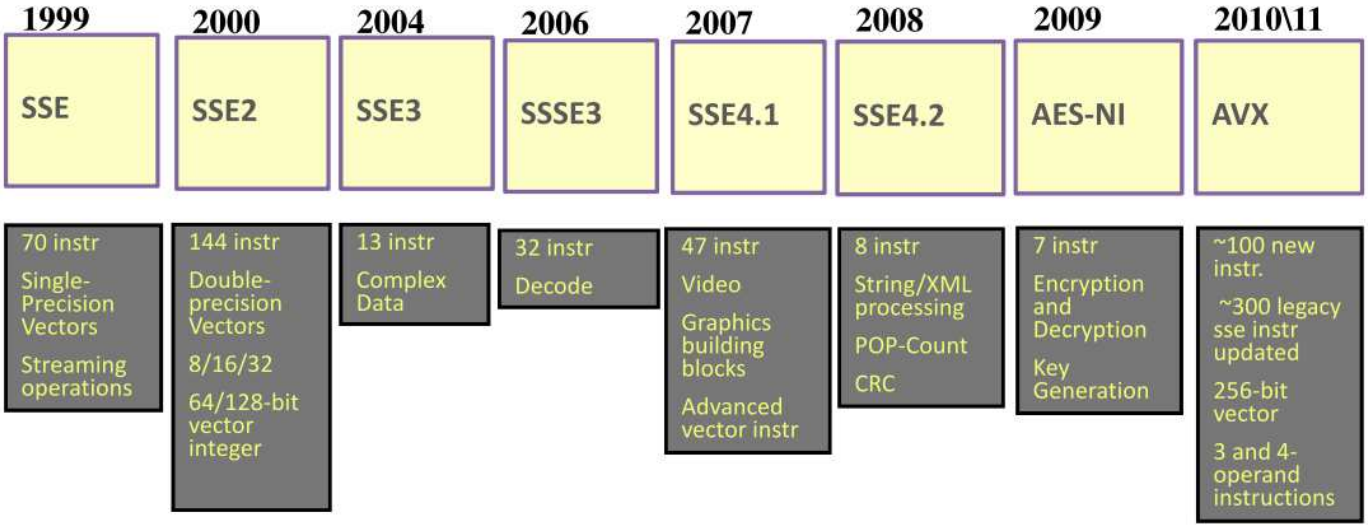
```
vector { unsigned | signed | bool } { char | short | long }

vector float
vector pixel
```



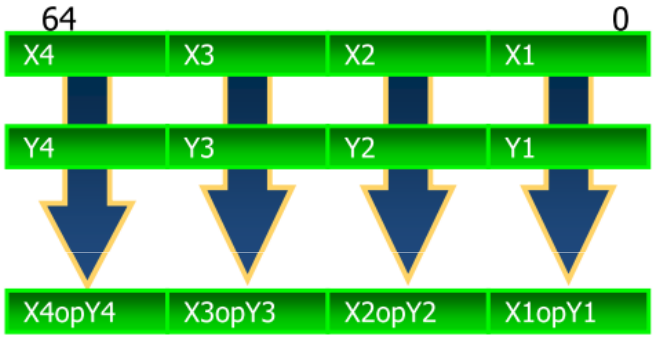
Three alternative vector register data layouts

Commercial SIMD architectures: Intel

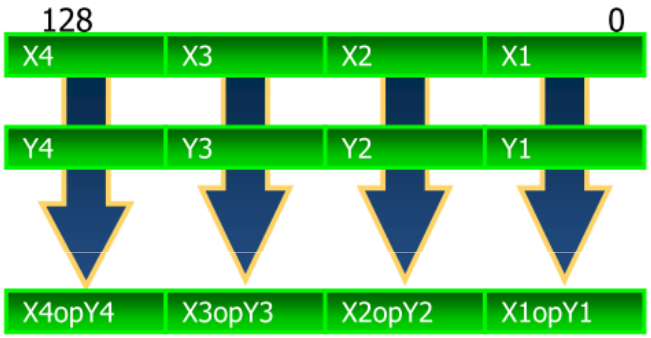
- Intel MMX (1996)
  - Repurpose 64-bit floating point registers
  - Eight 8-bit integer ops or four 16-bit integer ops
- Streaming SIMD Extensions (SSE) (1999)
  - Separate 128-bit registers: 8 16-bit ops, 4 32-bit ops, or two 64-bit ops
  - Single precision floating point arithmetic
- Double-precision floating point in
  - SSE2 (2001), SSE3(2004), SSE4(2007)
- Advanced Vector Extensions (2010)
  - Four 64-bit integer/fp ops



Commercial SIMD architectures: Intel

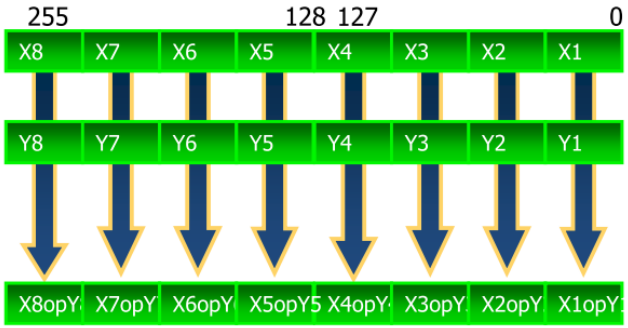


MMX™
Vector size: 64bit
Data types: 8, 16 and 32 bit integers
Vector Lane (VL): 2,4,8
For sample on the left: Xi, Yi 16 bit integers



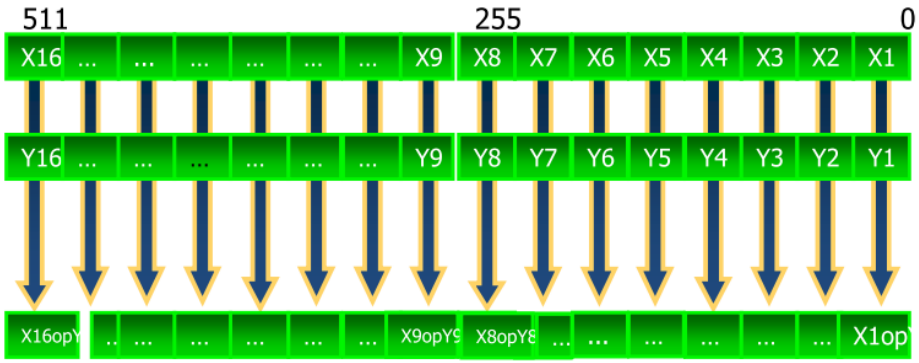
Intel® SSE
Vector size: 128bit
Data types:
8,16,32,64 bit integers
32 and 64bit floats
VL: 2,4,8,16
Sample: Xi, Yi bit 32 int / float

Commercial SIMD architectures: Intel



Intel® AVX
Vector size: 256 bit
(*Extendible to 512 and 1024 bits for future generations*)
Four 64-bit integer/fp ops
Data types: 32 and 64 bit floats
VL: 4, 8, 16
Sample: Xi, Yi 32 bit int or float

(Like the others, operands must be consecutive and aligned memory locations)



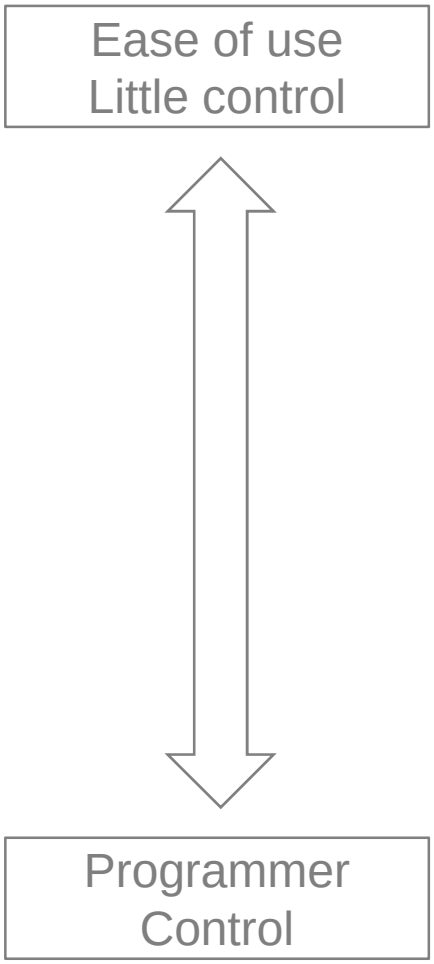
Intel® MIC
Vector size: 512bit
Data types:
32 and 64 bit integers
32 and 64bit floats
(some support for 16 bits floats)
VL: 8,16
Sample: 32 bit float

Programming with vector/SIMD extensions

- How to use SIMD (or other) machine instructions from high-level code?
- **`asm`**
  - inject assembly code from C/C++ sources
  - details depend on compiler implementation
  - some trickery to specify operands
- "intrinsics"
  - syntactically, look like function calls
  - each intrinsic directly corresponds to a specific processor instruction
  - again, details depend on compiler implementation

```
// Add 10 and 20 to register %eax
__asm__ ( "movl $10, %eax;"
          "movl $20, %ebx;"
          "addl %ebx, %eax;"   );
. . . . .

int no = 100, val ;
    asm ("movl %1, %%ebx;"
         "movl %%ebx, %0;"
         : "=r" ( val )   // output
         : "r" ( no )     // input
         : "%ebx" // clobbered register
     );
. . . . .

#include <xmmintrin.h>

__m128 source0 = {1.1, 2.2, 3.3, 4.4};
__m128 source1 = {10.1,20.2,30.3,40.4};
__m128 dest;  float
printarray[FLOAT_ARRAYSIZE]
        __attribute__ ((aligned (16)));
dest = _mm_shuffle_ps(source0, source1,
        _MM_SHUFFLE(0, 1, 2, 3));
```

Programming with vector/SIMD extensions



- Use Performance Libraries
- Compiler: Fully automatic vectorization
- Cilk Plus Array Notation
- Compiler: Auto vectorization hints
  (**`#pragma ivdep`**, …)
- User Mandated Vectorization (SIMD Directive)
- Manual CPU Dispatch
  (**`__declspec(cpu_dispatch …)`**)
- SIMD intrinsic class (**`F32vec4 add`**)
- Vector intrinsic (**`mm_add_ps()`**)
- Assembler code (**`addpsv`**)

Various Intel codes from a C routine

```
static double A[1000], B[1000],
              C[1000];
void add() {
    int i;
    for (i=0; i<1000; i++)
    if (A[i]>0)
    A[i] += B[i];
    else
    A[i] += C[i];
}
```

```
.B1.2::
movaps xmm2, A[rdx*8]
xorps xmm0, xmm0
cmpltpd xmm0, xmm2
movaps xmm1, B[rdx*8]
andps xmm1, xmm0
andnps xmm0, C[rdx*8]
orps xmm1, xmm0
addpd xmm2, xmm1
movaps A[rdx*8], xmm2
add       rdx, 2
cmp rdx, 1000
jl .B1.2
```

SSE2

```
.B1.2::
vmovaps   ymm3, A[rdx*8]
vmovaps   ymm1, C[rdx*8]
vcmpgtpd  ymm2, ymm3, ymm0
vblendvpd ymm4, ymm1,B[rdx*8], ymm2
vaddpd    ymm5, ymm3, ymm4
vmovaps   A[rdx*8], ymm5
add       rdx, 4
cmp       rdx, 1000
jl        .B1.2
```

AVX

```
.B1.2::
movaps    xmm2, A[rdx*8]
xorps     xmm0, xmm0
cmpltpd   xmm0, xmm2
movaps    xmm1, C[rdx*8]
blendvpd  xmm1, B[rdx*8], xmm0
addpd     xmm2, xmm1
movaps    A[rdx*8], xmm2
add       rdx, 2
cmp       rdx, 1000
jl        .B1.2
```

SSE4.1

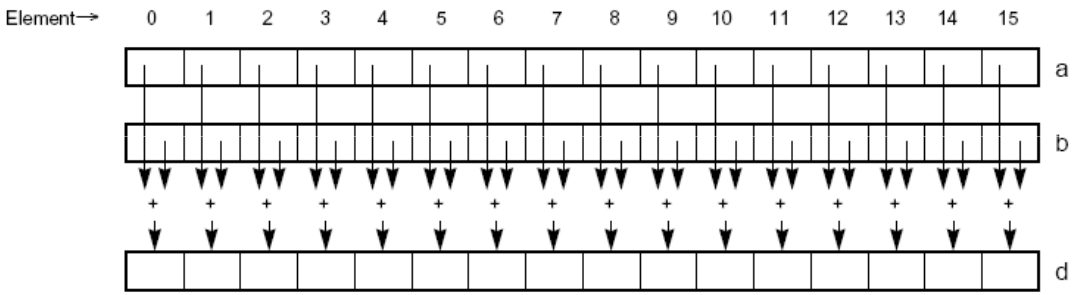
AltiVec Programming

- GNU Compiler Collection, IBM Visual Age Compiler and other compilers provide *intrinsics*
  - access AltiVec instructions directly from C and C++ programs
- The "**`vector`**" storage class is introduced
  - permits the declaration of native vector types
  - e.g., "**`vector unsigned char A`**;" declares a 128-bit vector variable named "**`A`**" containing sixteen 8-bit unsigned chars
- AltiVec C extensions map into AltiVec instructions
  - For example, **`vec_add()`** maps into one of four AltiVec instructions (**`vaddubm`**, **`vadduhm`**, **`vadduwm`**, or **`vaddfp`**) depending upon the types of the arguments to **`vec_add()`**.

AltiVec Programming: **vec_add**

- Example:
  - each element of **a** is added to the corresponding element of **b**
  - each sum is placed in the corresponding element of **d**

```
d = vec_add(a,b);
```



equivalent to:

Integer operands and addition:

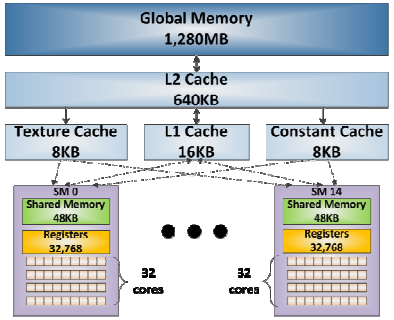
```
for(i=0; i<n; i++){
   d[i] = a[i]+b[i];
}
```

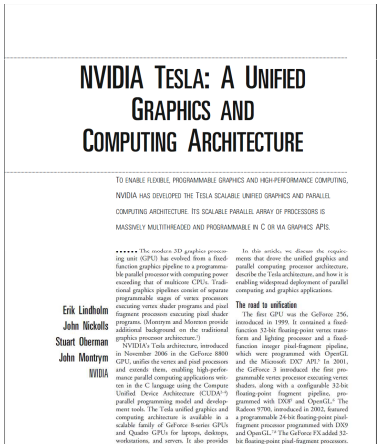
Floating point operands and addition:

```
for(i=0; i<4; i++){
   d[i] = a[i]+b[i];
}
```

GPU

- Graphics Processing Unit (GPU)
  - dedicated super-threaded, massively data parallel coprocessor
- Historically, followed a different path from SIMD
  - Hardware acceleration of dedicated graphics processing functions
- Now *General-Purpose* GPU (GPGPU)
  - General purpose programming model based on graphics-free API
  - Pushed by emerging general-purpose programming models:
    - NVIDIA's Compute Unified Device Architecture (**CUDA**), 2007
    - Khronos Group's Open Computing Language (**OpenCL**), 2009





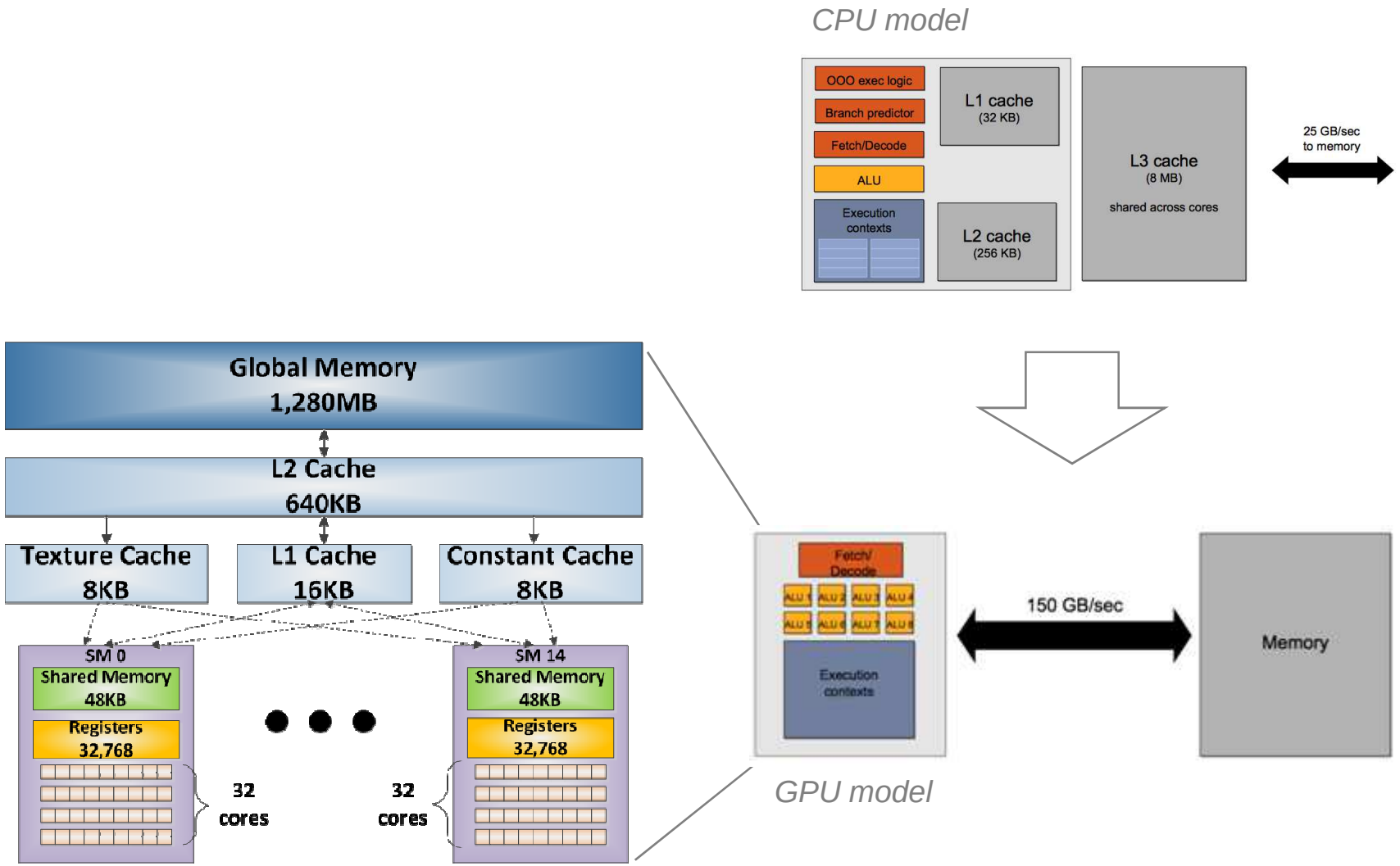
NVIDIA TESLA: A UNIFIED GRAPHICS AND COMPUTING ARCHITECTURE

TO ENABLE FLEXIBLE, PROGRAMMABLE GRAPHICS AND HIGH-PERFORMANCE COMPUTING, NVIDIA HAS DEVELOPED THE TESLA SCALABLE UNIFIED GRAPHICS AND PARALLEL COMPUTING ARCHITECTURE. ITS SCALABLE PARALLEL ARRAY OF PROCESSORS IS MASSIVELY MULTITHREADED AND PROGRAMMABLE IN C OR VIA GRAPHICS APIS.

Erik Lindholm
John Nickolls
Stuart Oberman
John Montrym
NVIDIA

The 2008 IEEE paper presenting Tesla. "Tesla" is today the code name of a family of products including devices like K20, K20X, also used in HPC

Current GPGPUs

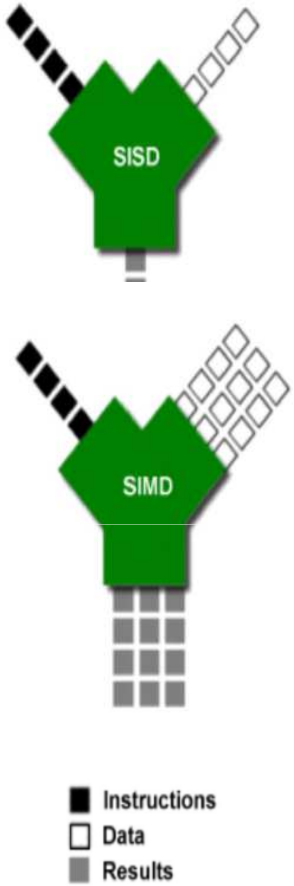


Current GPGPUs

- The GPU is viewed as a compute device that:
  - Is a co-processor to the CPU or host
  - Has its own DRAM (global memory in CUDA parlance)
  - Runs many threads in parallel
- Data-parallel portions of an application run on the device as kernels which are executed in parallel by many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few heavy ones

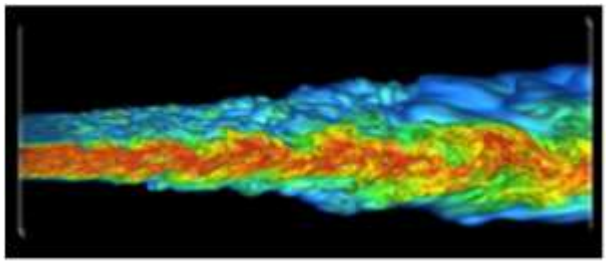
Current GPGPUs

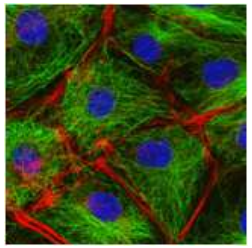
- GPU is a in fact SIMD device → works on "streams" of data
  - Each "GPU thread" executes one general instruction on the stream of data that the GPU is assigned to process
  - sometimes called *single instruction multiple thread* (SIMT)
- Compute power comes from a vertical hierarchy:
  - e.g., NVIDIA: set of Streaming Multiprocessors (SMs)
  - each SM has a set of 32 Scalar Processors (SPs)
    - Maxwell has 128 SPs, Kepler has 196 SPs, Fermi 2.1 had 48 SPs
- The quantum of scalability is the SM
  - larger (and more expensive) GPUs have more SMs
  - Fermi can have up to 16 SMs on one GPU card



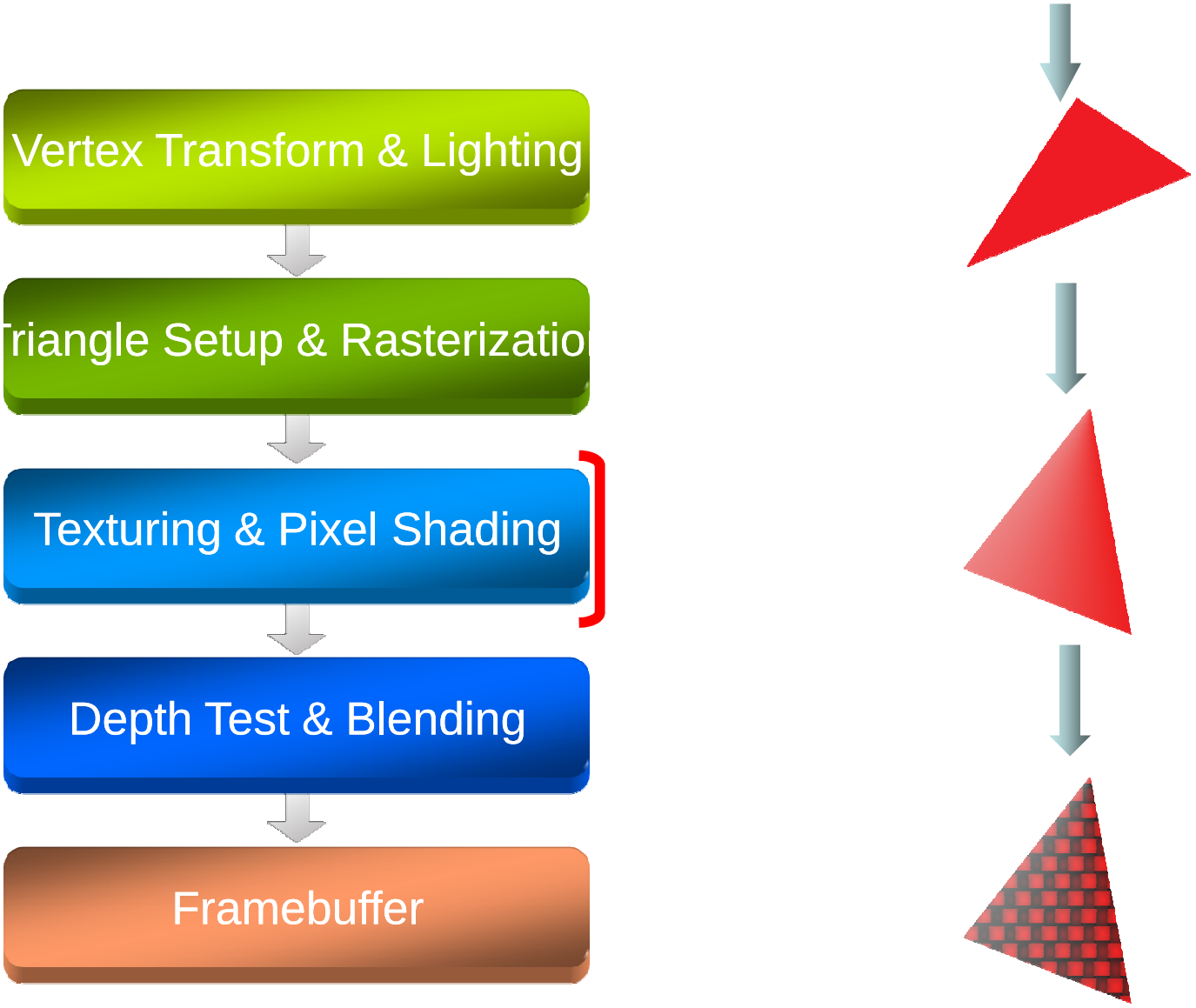
Graphics computing

- Workload and Programming Model provide lots of parallelism:
  - Graphics applications provide large groups of vertices at once
  - Vertices can be processed in parallel
  - Apply the same transform to all vertices
- Triangles contain many pixels
  - Pixels from a triangle can be processed in parallel
  - Apply the same *shader* to all pixels
- Very efficient hardware to hide serialization bottlenecks



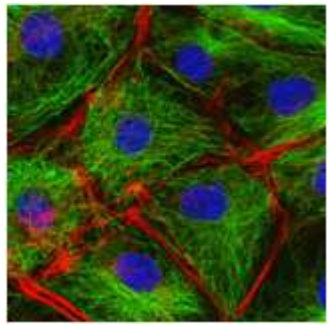


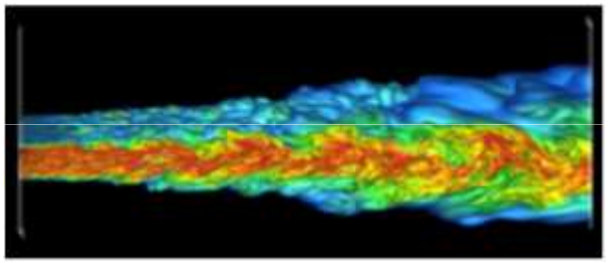
The Graphics Pipeline



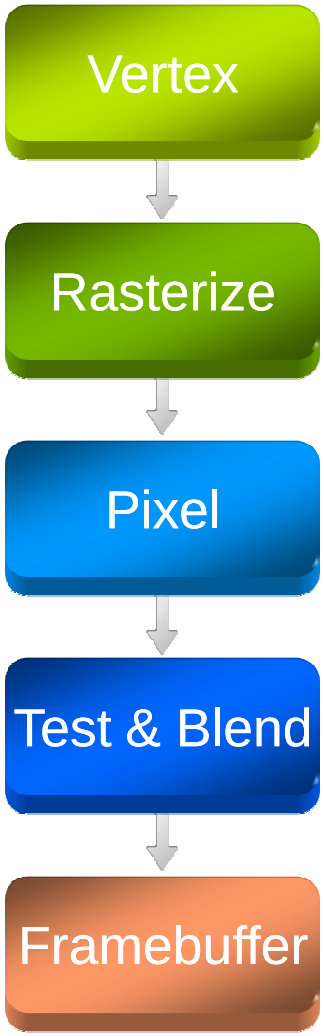
Applications of Computer Graphics

- Climate
  - E.g.: Comprehensive Earth System Model at 1KM scale, enabling modeling of cloud convection and ocean eddies
- Biology
  - E.g.: Coupled simulation of entire cells at molecular, genetic, chemical and biological levels
- Astrophysics
  - E.g.: Predictive calculations for thermonuclear and core-collapse supernovae, allowing confirmation of theoretical models.
- Gaming applications
- etc…





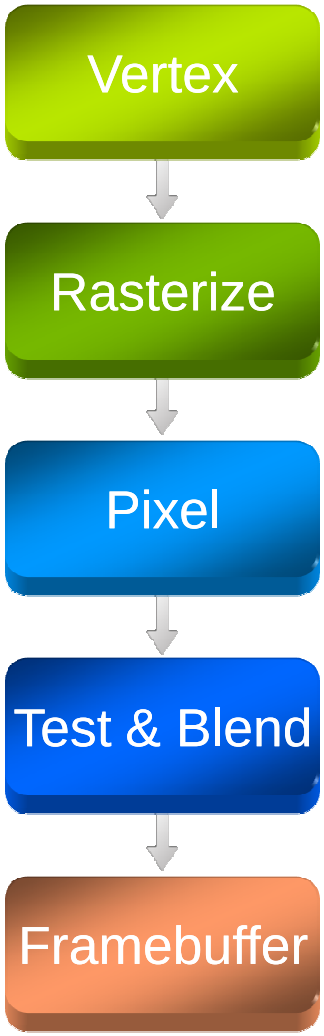
Evolution of GPUs



Early Graphics accelerators:

- Key abstraction of real-time graphics
- Hardware directly resembled the pipeline structure
- One chip/board per stage
- Fixed data flow through pipeline

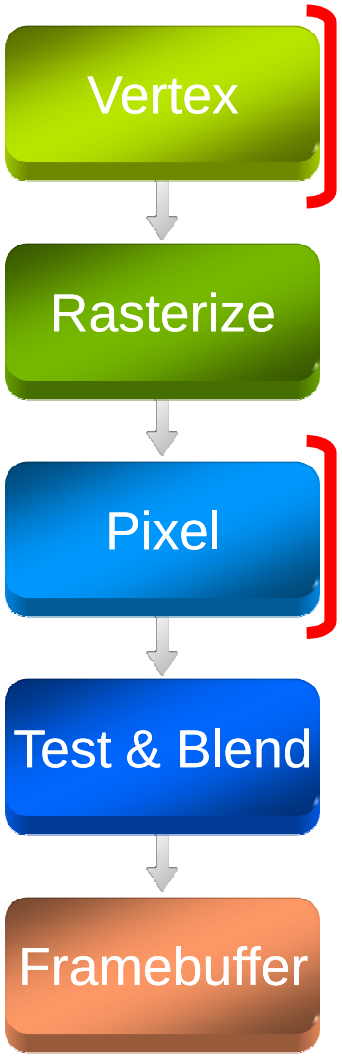
Evolution of GPUs



Early GPUs:

- Everything was a fixed function, with a certain number of modes
- Number of modes for each stage grew over time
- Hard to optimize hardware
- Developers wanted more flexibility

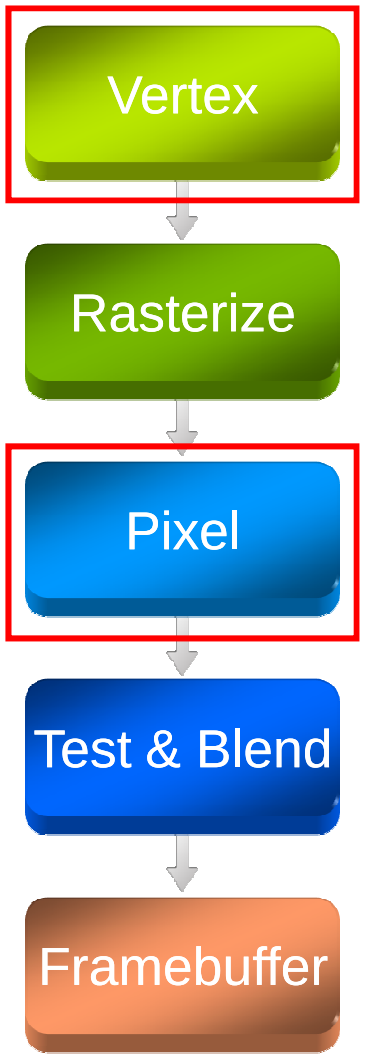
Evolution of GPUs



Programmable GPUs:

- Remains a key abstraction
- Hardware still directly resembled the pipeline structure
- but, Vertex & Pixel processing became programmable
  - new stages added
- GPU architecture increasingly centers around *shader* execution

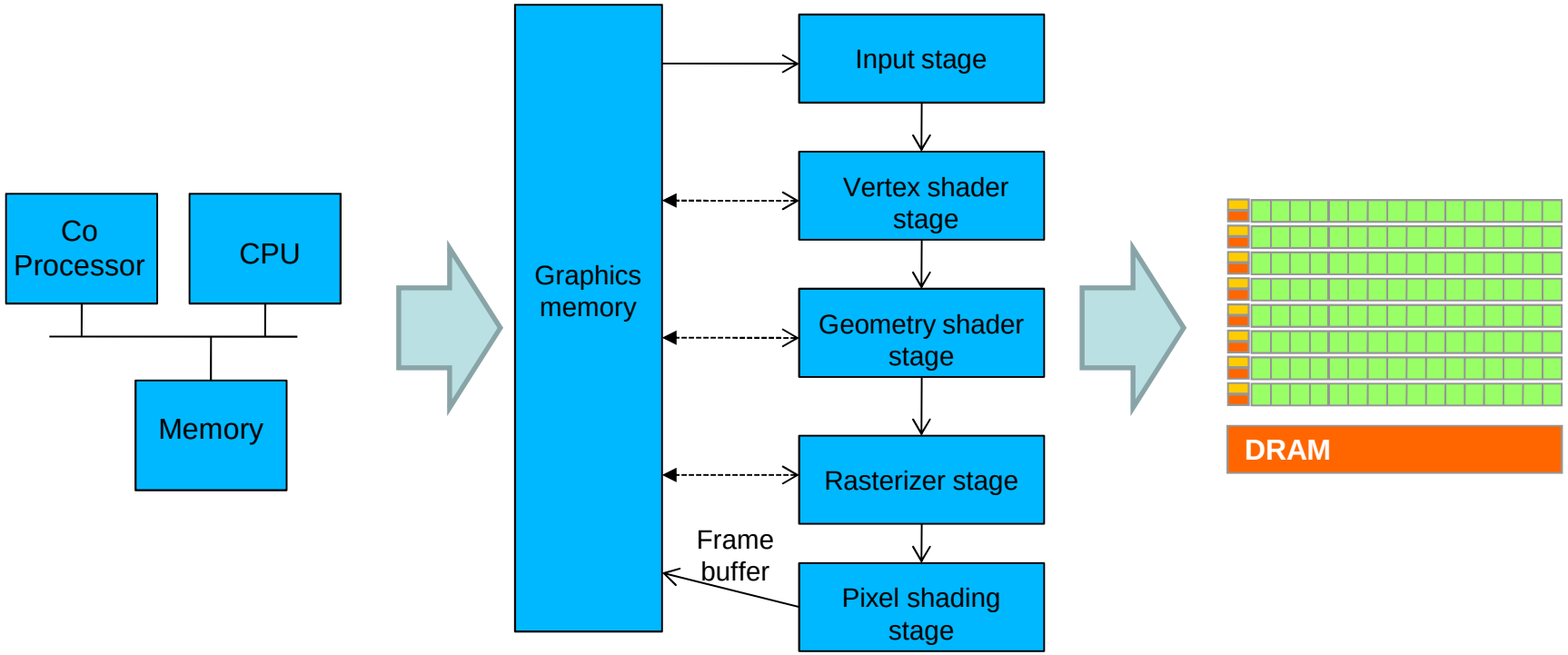
Evolution of GPUs



Programmable GPUs:

- Exposing a (initially limited) instruction set for some stages
- Limited instructions & instruction types and no control flow at first
- Then expanded to full ISA

Evolution of GPU architectures

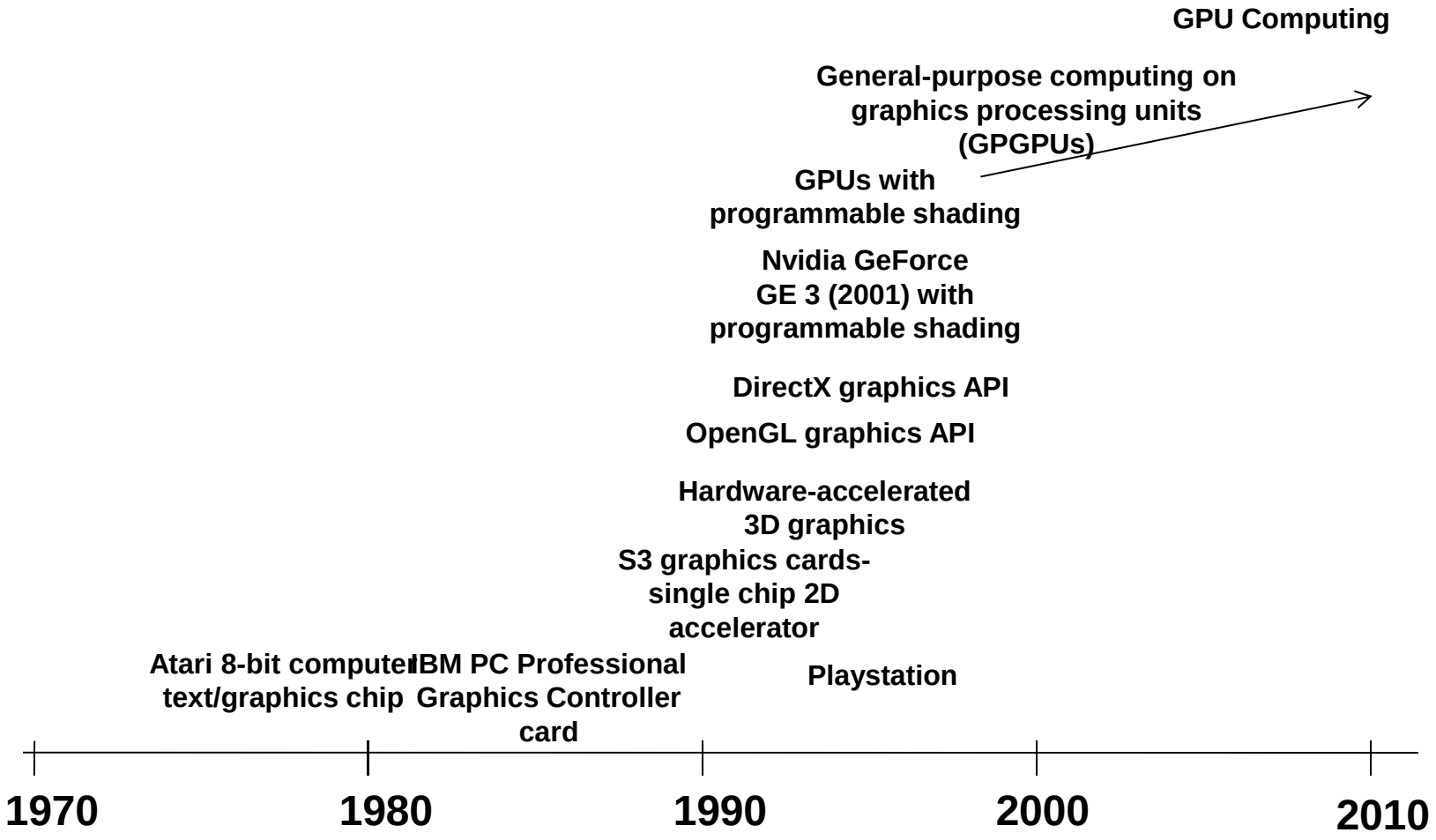


1970-1980: floating point co-processors attached to microprocessors

Late 1990s: graphics chips needed to support 3-D (graphics APIs such as DirectX and OpenGL)
Graphics chips generally had a pipeline structure: a sequence of highly specialized operations

Mid 2000s: General-Purpose GPUs (GPGPUs)

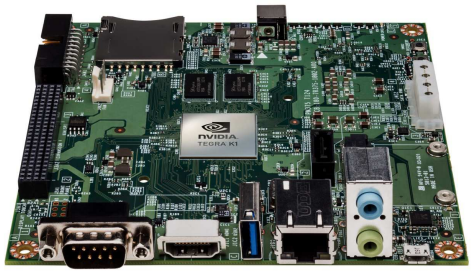
Evolution of GPUs



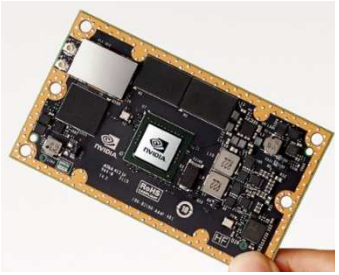
GPU acceleration: New areas

- Embedded/Automotive
- “Infotainment”
- etc…

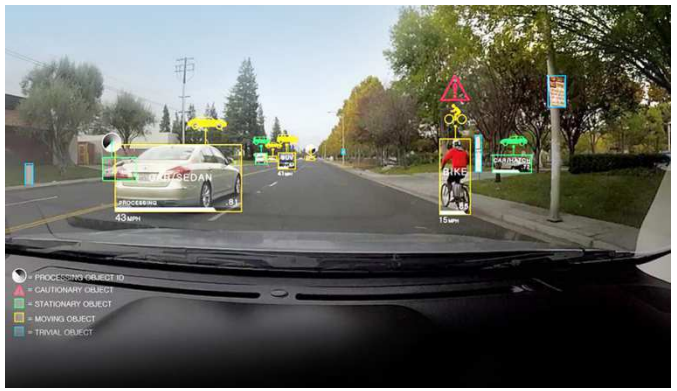
- NVIDIA PX
  - Self Driving Car Computing
- NVIDIA CX
  - Digital Cockpit Computer



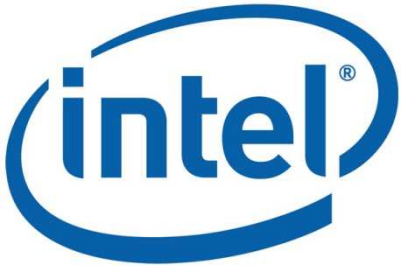
NVIDIA Jetson TK1

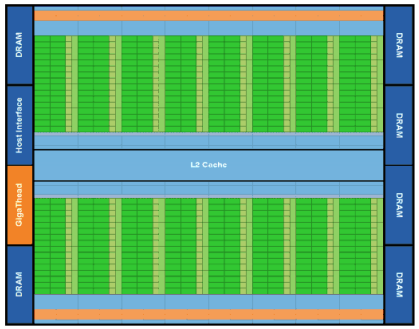


NVIDIA Jetson TX1

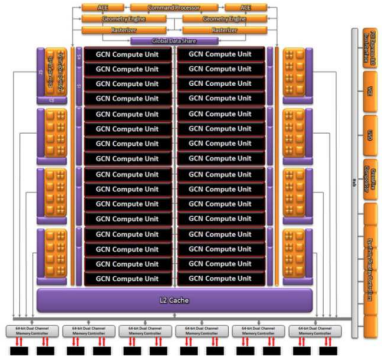


GPGPU: main players

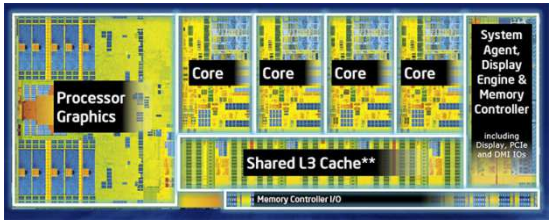




(NVIDIA Fermi Architecture)



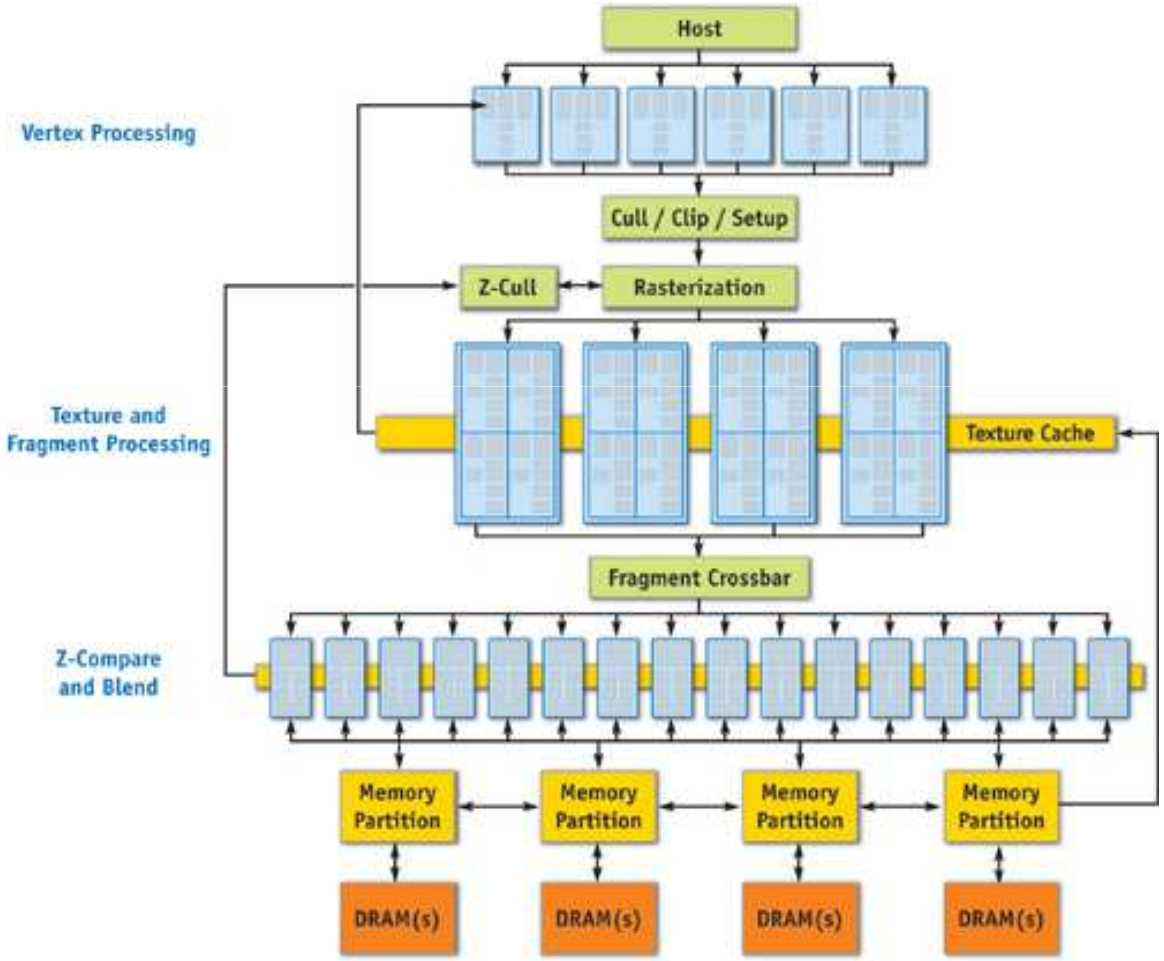
(AMD Tahiti Architecture)



(Intel i7 with Processor Graphics)

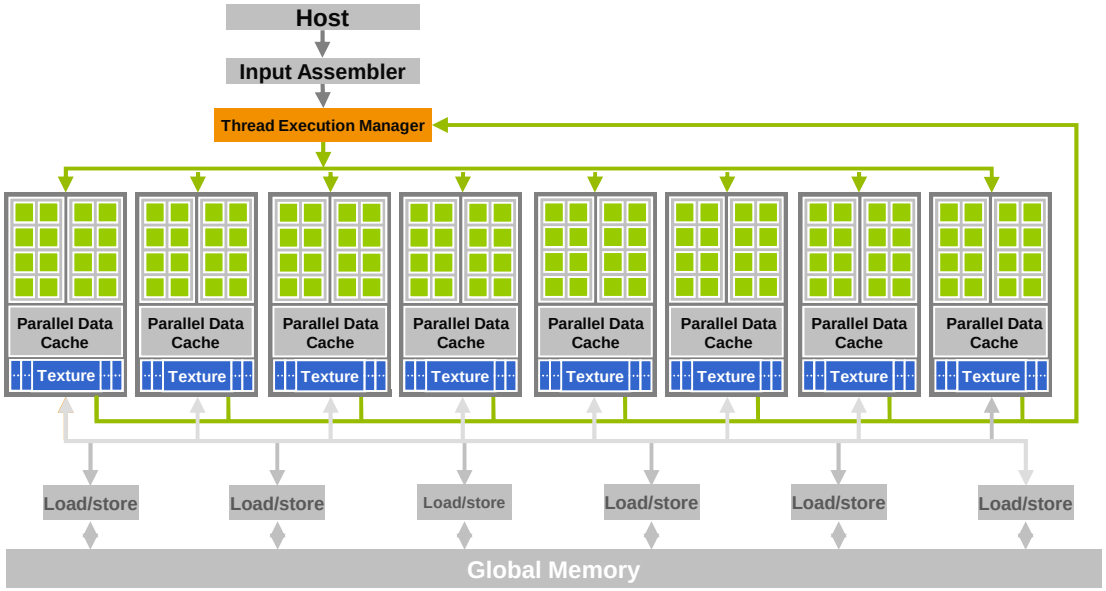
GeForce 6 Series Architecture (2004-5)

- Fixed-pipeline architecture
  - still no GPGPU
- Compliant with Microsoft DirectX 9.0c specification and OpenGL 2.0



NVIDIA G80 chip/GeForce 8800 card (2006)

- First GPU for HPC as well as graphics
  - unified processors could perform vertex, geometry, pixel, and general computing operations
- Could now write programs in C rather than graphics APIs
- Single-instruction multiple thread (SIMT) programming model

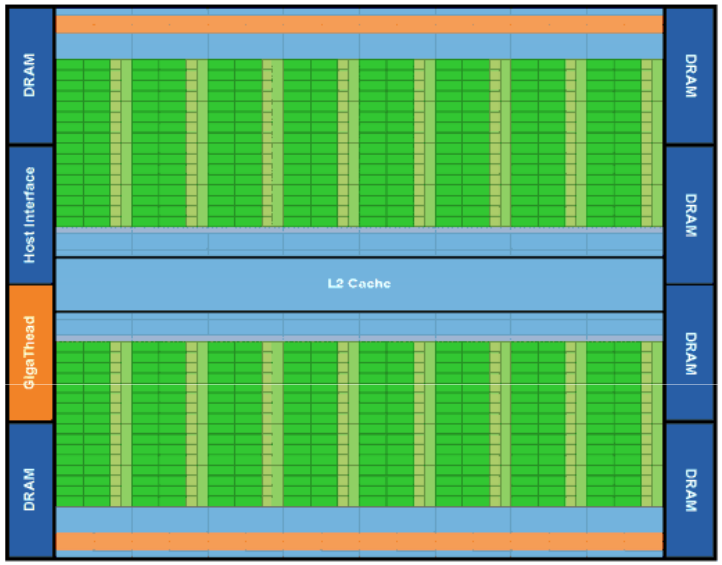


G80 Device:

- Processors execute computing threads
- Thread Execution Manager issues threads
- 128 Thread Processors grouped into **16 multiprocessors** (SMs)
- Parallel Data Cache enables thread cooperation

NVIDIA Fermi architecture (Sept 2009)

- First implementation: Tesla 20 series
- 16 Streaming Multiprocessors (**SM**s)
  - each having 32 stream processing engines (**SPE**s): up to 512 cores
- Many innovations
  - e.g., L1/L2 caches, unified device memory addressing, ECC memory,…
- ~3 billion transistor chip
- Number of cores limited by power considerations
  - e.g. C2050 has 448 cores



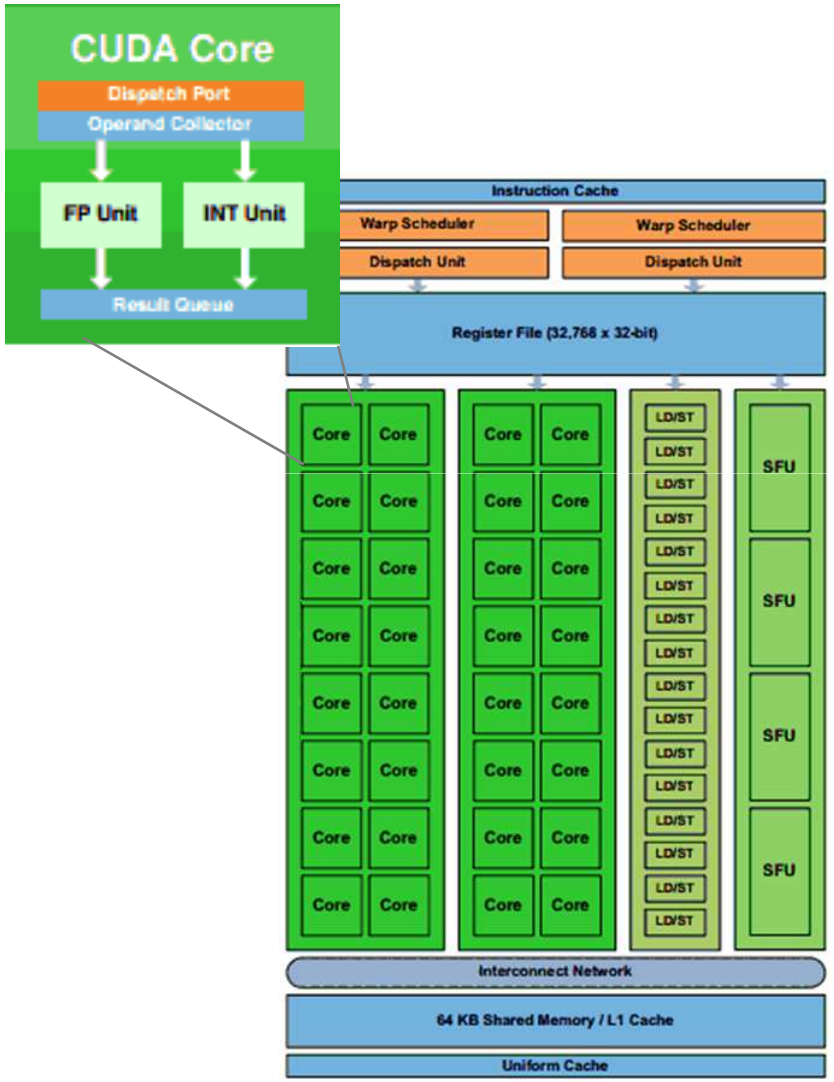
Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

GF100: NVIDIA Fermi

- GPU/CPU Host interface: PCI-Express v2 bus (peak transfer rate of 8GB/s)
- DRAM: up to 6GB of GDDR5 DRAM, 64-bit addressing capability
- Clock frequency: ~1.5GHz, Peak performance: 1.5 TFlops.
- DRAM bandwidth: 192GB/s

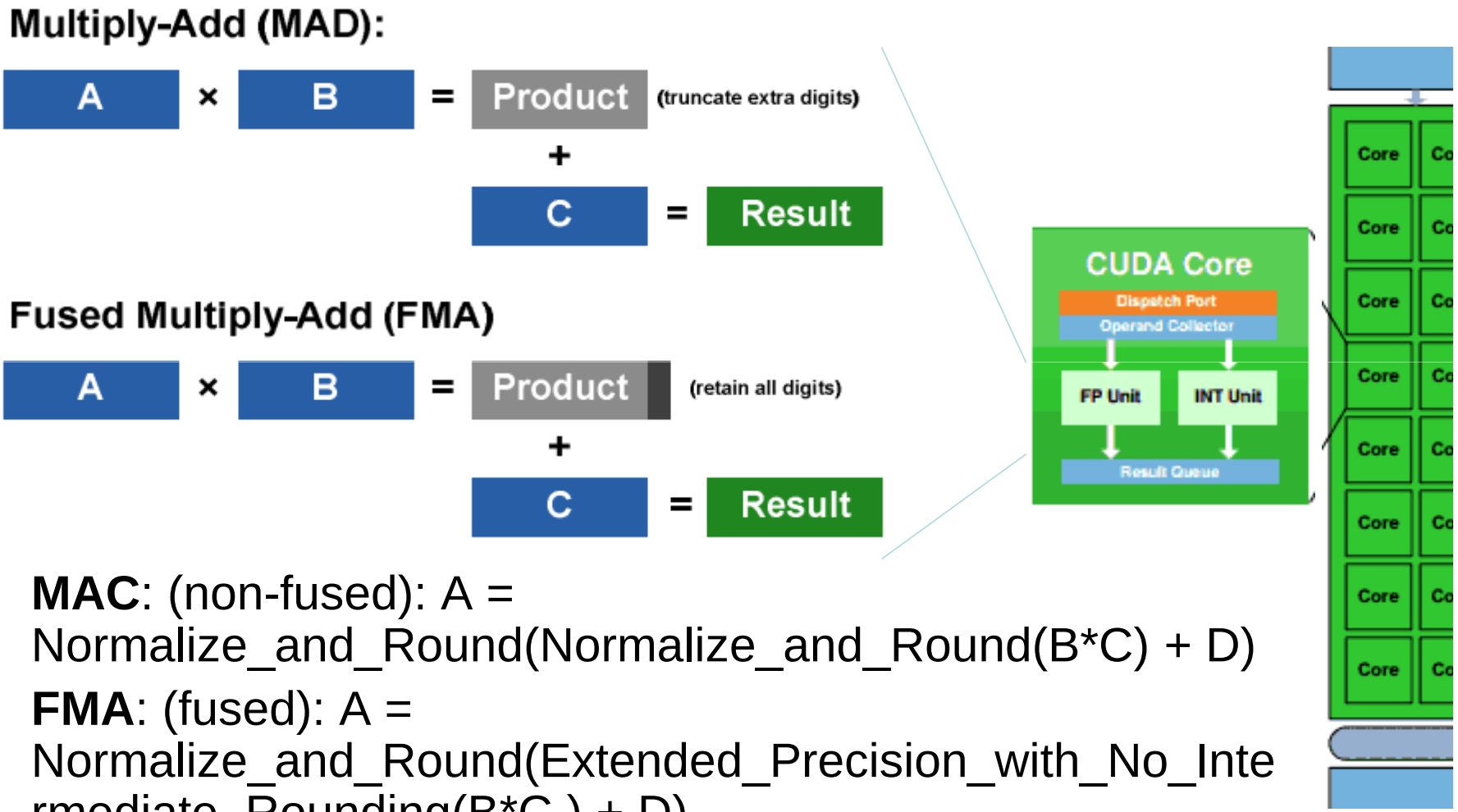
Fermi Streaming Multiprocessor (SM)

- 32 CUDA cores
  - Fully pipelined Integer ALU and FPU
  - 32-bit Integer operations
  - IEEE-754:2008 Double Precision Operations
  - Support for: Boolean, shift, move, compare, convert, bit-field extract, bit-reverse insert
- 32K of 32-bit register file
- 64 KB of RAM with a configurable partitioning of shared memory and L1 cache 48:16 or 16:48
- 16 Load/Store units
  - source and destination addresses to be calculated for 16 threads per clock
  - Load and store the data from/to cache or DRAM
- 4 Special Function Unit (SFU) for transcendental instructions:
  - e.g. sin, cosine, reciprocal, and square root
  - Each SFU executes one instruction per thread, per clock; a warp executes over eight clocks.
  - The SFU pipeline is decoupled from the dispatch unit, allowing the dispatch unit to issue to other execution units while the SFU is occupied.
- Dual Warp Scheduler simultaneously schedules and dispatches instructions from two independent *warps*
- GF104/GF114 are also 16 CUDA cores FP64 capable, 8 SFU and 4 dispatched units*



Fermi Streaming Multiprocessor (SM)

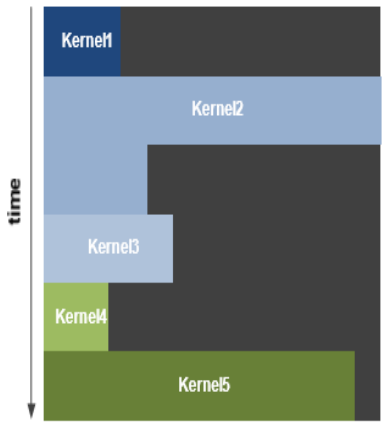
Fused MultiplyAdd (FMA)



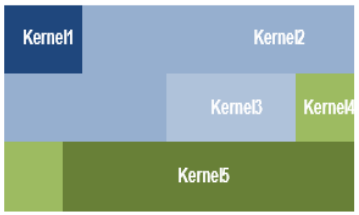
- **MAC**: (non-fused): A = Normalize_and_Round(Normalize_and_Round(B*C) + D)
- **FMA**: (fused): A = Normalize_and_Round(Extended_Precision_with_No_Intermediate_Rounding(B*C) + D)

The Fermi GigaThread™ Thread Scheduler

- Global GPU scheduler:
- Schedules thread blocks to various SMs
  - Check for resources availability
  - Example:
    - each thread can use only 63 registers
    - Only 16 threadblocks per SM
- 10x faster application context switching (Compared to G80)
- Concurrent kernel execution
- Out of Order thread block execution
- Dual overlapped memory transfer engines



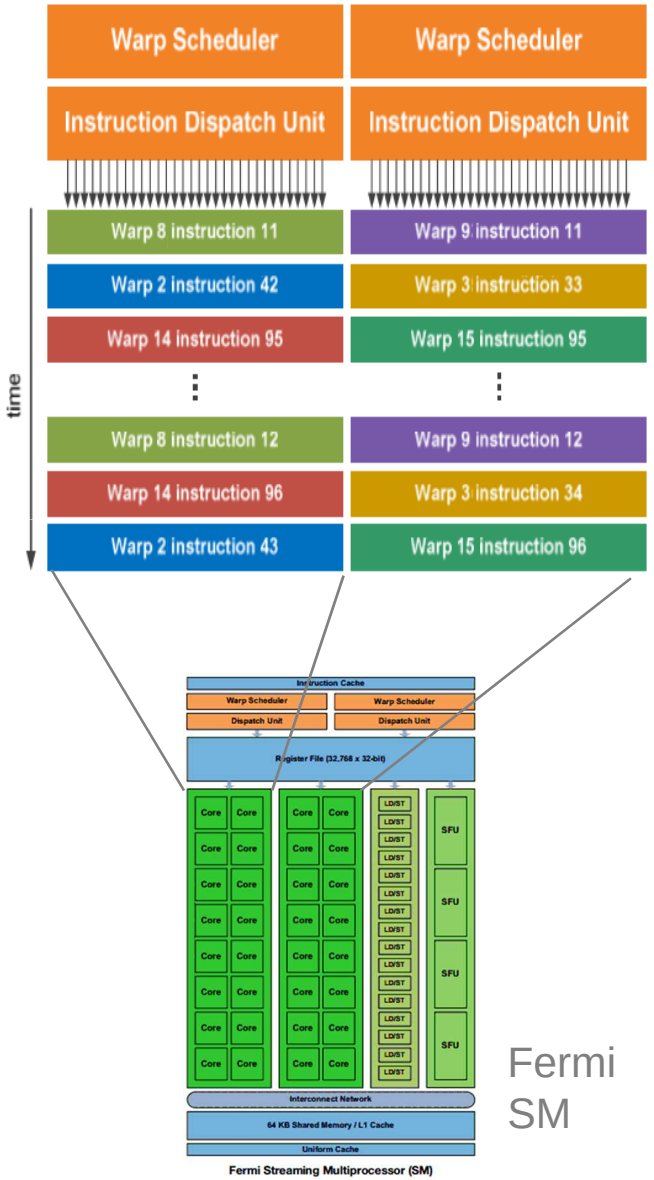
Serial Kernel Execution



Concurrent Kernel Execution

The Fermi *Dual Warp Scheduler*

- "Local" SM scheduler
  - handles groups of 32 parallel threads called *warps*
- Each SM has two warp schedulers and two instruction dispatch units
  - allow two warps to be issued and executed concurrently
- Selects two warps and issues one instruction from each warp to a SM part, i.e. a group of 16 cores, 16 load/store units, or 4 SFUs
  - most instructions can be dual issued (exceptions include double precision instructions, which cannot be dual issued)
  - efficient scheduling allow near-peak HW performance
- Because warps execute independently, Fermi's scheduler does not need to check for dependencies within the instruction stream
  - however, the architecture with four dispatcher units do have a check for dependencies
- Max 48 Warps per SM



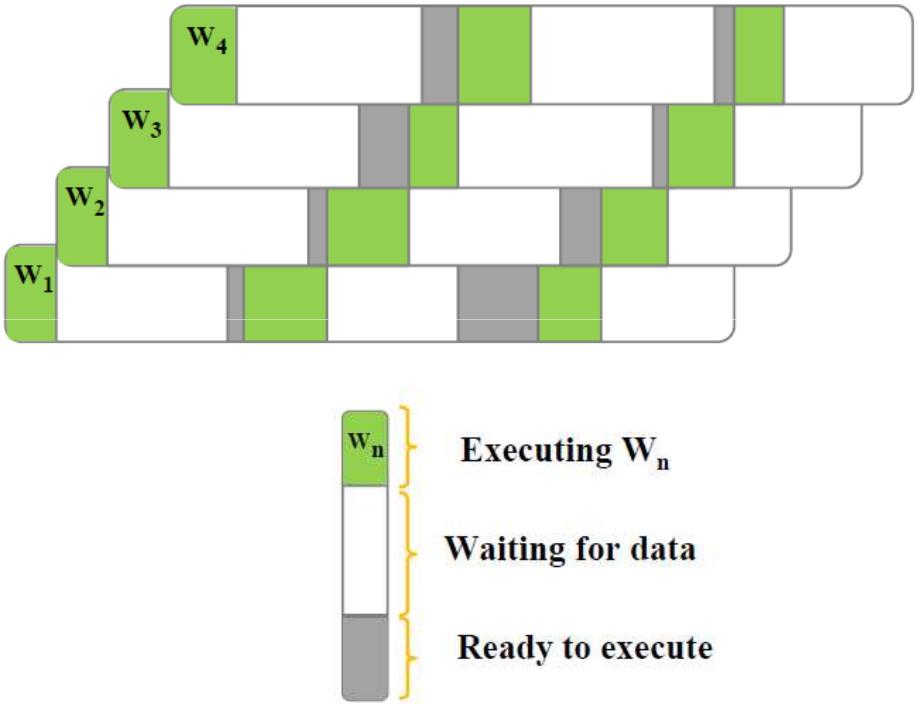
Fermi SM

Fermi Shader Clock

- On GF100/104/110/114:
  - Within the SM itself different units operated on *different* clocks
    - schedulers and texture units operating on the core clock (607 MHz)
    - CUDA cores, load/store units, and SFUs operated on the shader clock, which ran at twice the core clock (1215 MHz).
  - With Fermi, a warp would be split up and executed over 2 cycles of the shader clock;
    - 16 threads would go first, and then the other 16 threads over the next clock.
    - The shader clock allows a full warp to be executed over a single graphics clock cycle (at 607 MHz) while only using enough hardware for half of a warp
    - *Half Warp* is the true working unit

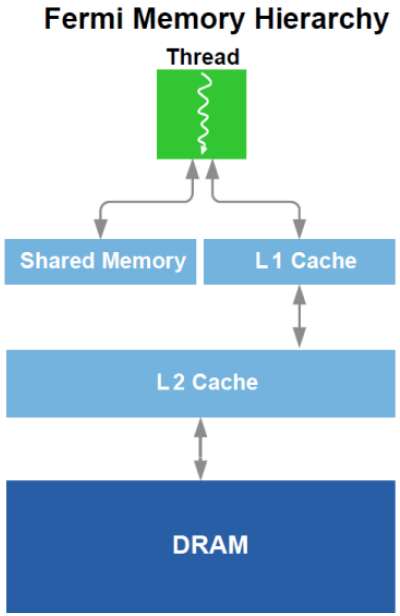
Thread and Warp Scheduling

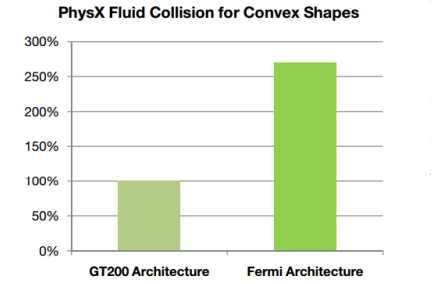
- An SM can switch between warps with no overhead
  - warps with instruction whose inputs are ready are eligible to execute, and will be considered when scheduling
  - When a warp is selected for execution, all active threads execute the same instruction in lockstep fashion
- Applies to both Fermi and Kepler



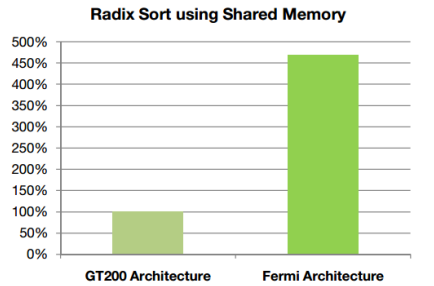
The Fermi Memory Subsystem

- One L1 cache per SM multiprocessor and unified L2 cache that services all operations (load, store and texture)
- 768 KB of L2 Cache
- The per-SM L1 cache is configurable to support both shared memory and caching of local and global memory operations:
  - the 64 KB memory can be configured as either 48 KB of Shared memory with 16 KB of L1 cache, or 16 KB of Shared memory with 48 KB of L1 cache
- ECC Memory Support
- Fast Atomic Memory Operations
  - Performance is up 20x faster compared to Tesla





Physics algorithms such as fluid simulations especially benefit from Fermi's caches. For convex shape collisions, Fermi is 2.7x faster than GT200.



When using 48 KB of shared memory on Fermi, Radix Sort executes 4.7x faster than GT200.

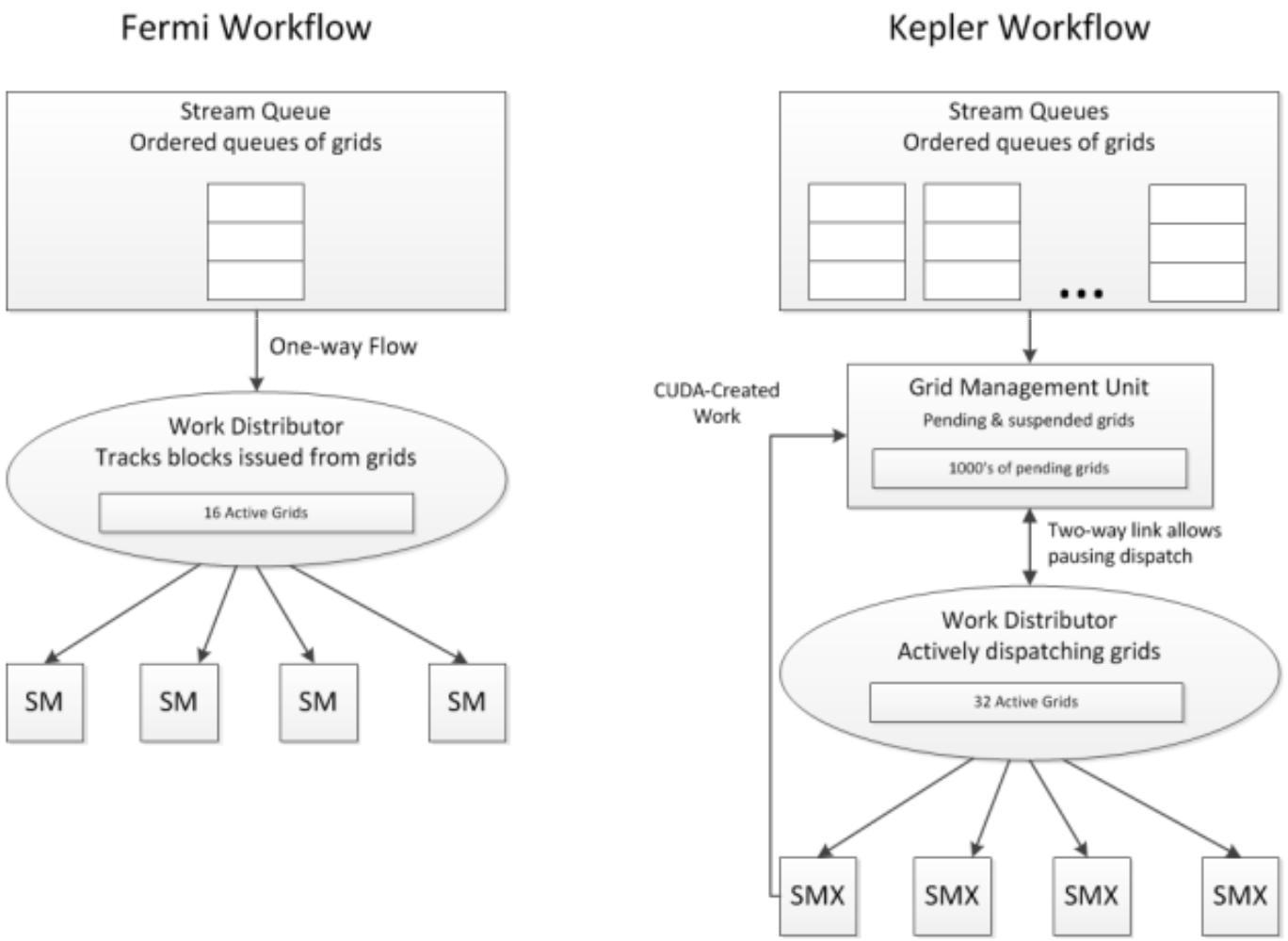
The Kepler Architecture

- A representative device: GK110 (2013)
  - CUDA Computer Capability 3.5
  - 7.1 billion transistors, 28nm manufacturing process
  - 3x Performance per Watt on Fermi
- Drop Shader Clock, Doubling Resources
  - One Clock (~700MHz) = Power efficiency
- Streaming Multiprocessor (SMX)
  - 15 SMX each composed by 192 CUDA cores
- GigaThread globlal scheduler
  - distributes thread blocks to SM thread schedulers and manages the context switches between threads during execution
- Host interface
  - GPU-CPU connected via a PCI-Express v3 bus (peak transfer rate: 16 GB/s).
- DRAM
  - six 64-bit memory controllers
- L2 cache: 2x Capacity compared to Fermi
- Example of other devices:
  - GK104 GPU, GTX 680 card: 1536 cores, 195 watts (March 2012)
  - GXT 690 has two dies, 3072 cores (2 x 1536 cores), 300 watts (April 2012)
  - CUDA Computer Capability 3.0

Kepler new features

- Dynamic Parallelism
  - GPU can generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths, without involving the CPU
- Hyper-Q
  - multiple CPU cores can launch work on a single GPU simultaneously, increasing GPU utilization and reducing CPU idle times
  - 32 simultaneous, hardware-managed connections (work queues) between the host and the GPU (Fermi has only a single connection)
- Grid Management Unit
  - Dynamic Parallelism requires an advanced grid management and dispatch control system
  - The Grid Management Unit (GMU) manages and prioritizes grids to be executed
- NVIDIA GPUDirect™
  - multiple GPUs within a single computer, or even in different servers across a network, can directly exchange data without needing to go to CPU/system memory
- Shuffle Instruction
  - Kepler implements a new Shuffle instruction which allows threads within a warp to share data *without using shared memory*, e.g.: **`__shf()`**, **`__shfl_up()`**, **`__shfl_down()`**, **`__shfl_xor()`**

Kepler Grid Management Unit

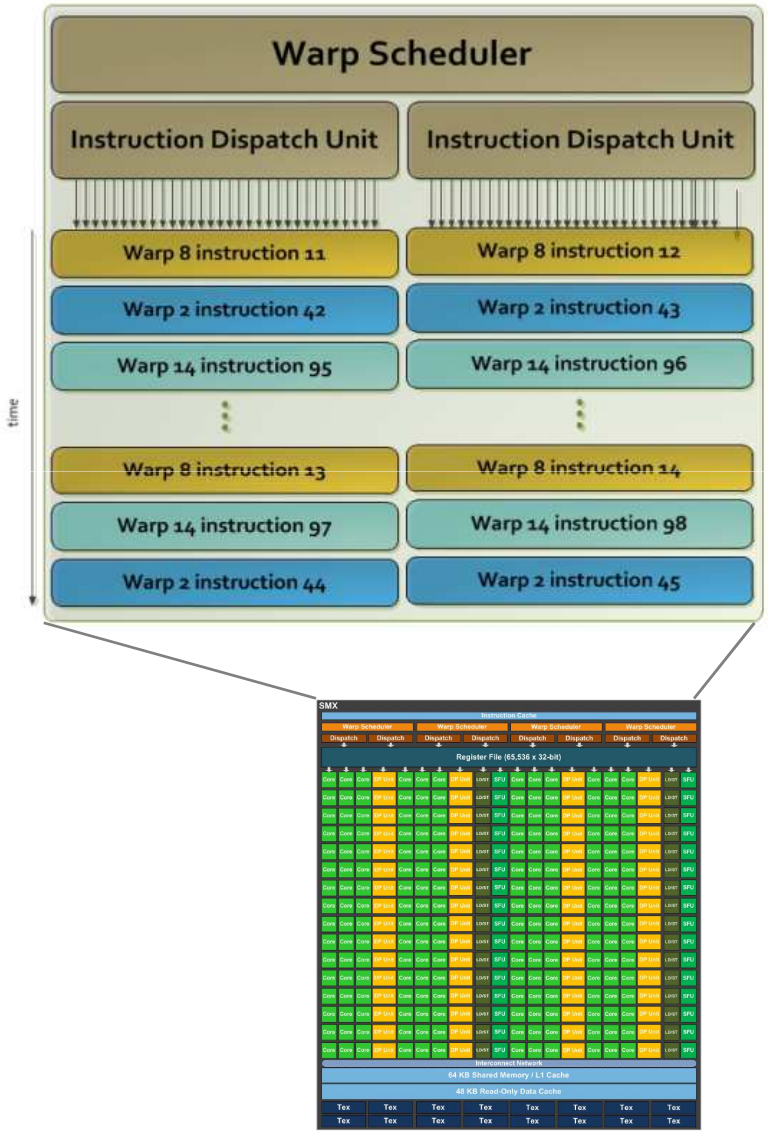


Kepler Streaming Multiprocessor (SMX)

- 192 single precision CUDA cores
  - Fully pipelined Integer ALU and FPU
  - 32-bit Integer operations
- 64 double precision units
  - not present in all Kepler GK*xxx*)
  - IEEE-754:2008 Single and Double Precision
- 32 Load/Store units
  - source and destination addresses can be calculated for 32 threads per clock
  - Load and store the data from/to cache or DRAM
- 32 Special Function Unit
  - transcendental instructions such as sin, cosine, reciprocal, and square root
  - SFU pipeline is decoupled from dispatcher → can issue to other execution units while the SFU is occupied
- 64K of 32-bit register file
- A 4-Warp Scheduler simultaneously schedules and dispatches instructions through 8 dispatch units
- 64 KB of RAM
  - configurable partitioning of shared memory and L1 cache (48/16 or 16/48 or 32/32 KB)
- 48 KB Read-Only Data Cache

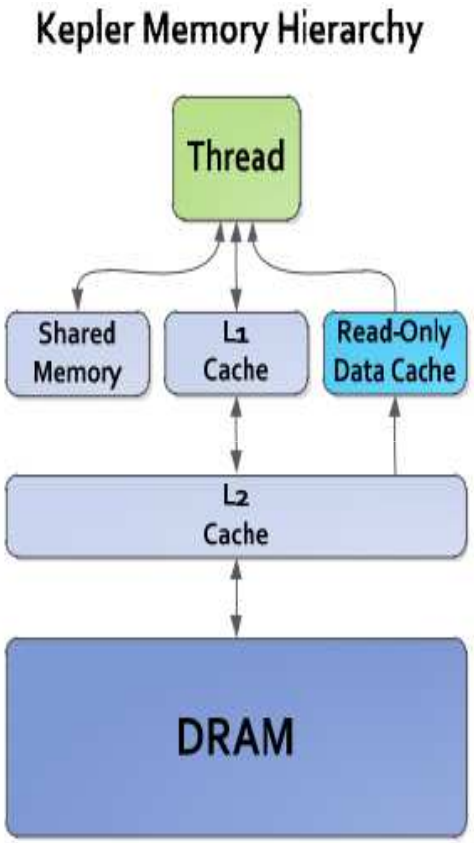
Kepler Quad Warp Scheduler

- handles groups of 32 parallel threads (*warps*)
- Each SMX features 4-warp schedulers and 8 instruction dispatch units
  - allows 4 warps to be issued and executed concurrently
  - selects 4 warps, and 2 independent instructions per warp can be dispatched each cycle
- Uses deterministic information (e.g. Math pipeline latencies) to resolve data hazard
  - remove Fermi complex hardware
  - improved power efficiency
- Kepler GK110 allows double precision instructions to be paired with other instructions
  - unlike Fermi, which only permits single issue of DP instructions



Kepler Memory Subsystem hints

- Single unified memory request path for load and store operations
- One L1 cache per SM multiprocessor
- Unified 1536 KB L2 cache serving all operations
  - load, store and texture
- The per-SM L1 cache is configurable to support both shared memory and local/global memory caching
  - The 64 KB memory can be configured as either 48 KB of Shared memory with 16 KB of L1 cache, or 16 KB / 48 KB, or 32 KB / 32 KB
- 48 KB Read-Only Data Cache
  - In Fermi it was accessible only by the Texture unit
- ECC Memory Support
  - Read-Only Data Cache supports single-error correction through a parity check
  - Note: ECC consumes bandwidth
- Fast Atomic Memory Operations
  - Same performance of Fermi but more operations are supported



Kepler vs. Fermi

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

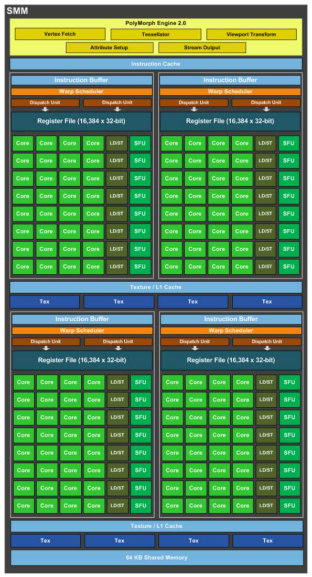
Compute Capability of Fermi and Kepler GPUs

NVIDIA Maxwell architecture and GPUs (2014)

- First used in GeForce GTX 750 and the GeForce GTX 750
- new design for the Streaming Multiprocessor (SM)
- considerable improvements in power efficiency
- increased the amount of L2 cache from 256 KB on GK107 to 2 MB on GM107
- cut the memory bus from 192 bit on GK106 to 128 bit on GM107 (for power saving)

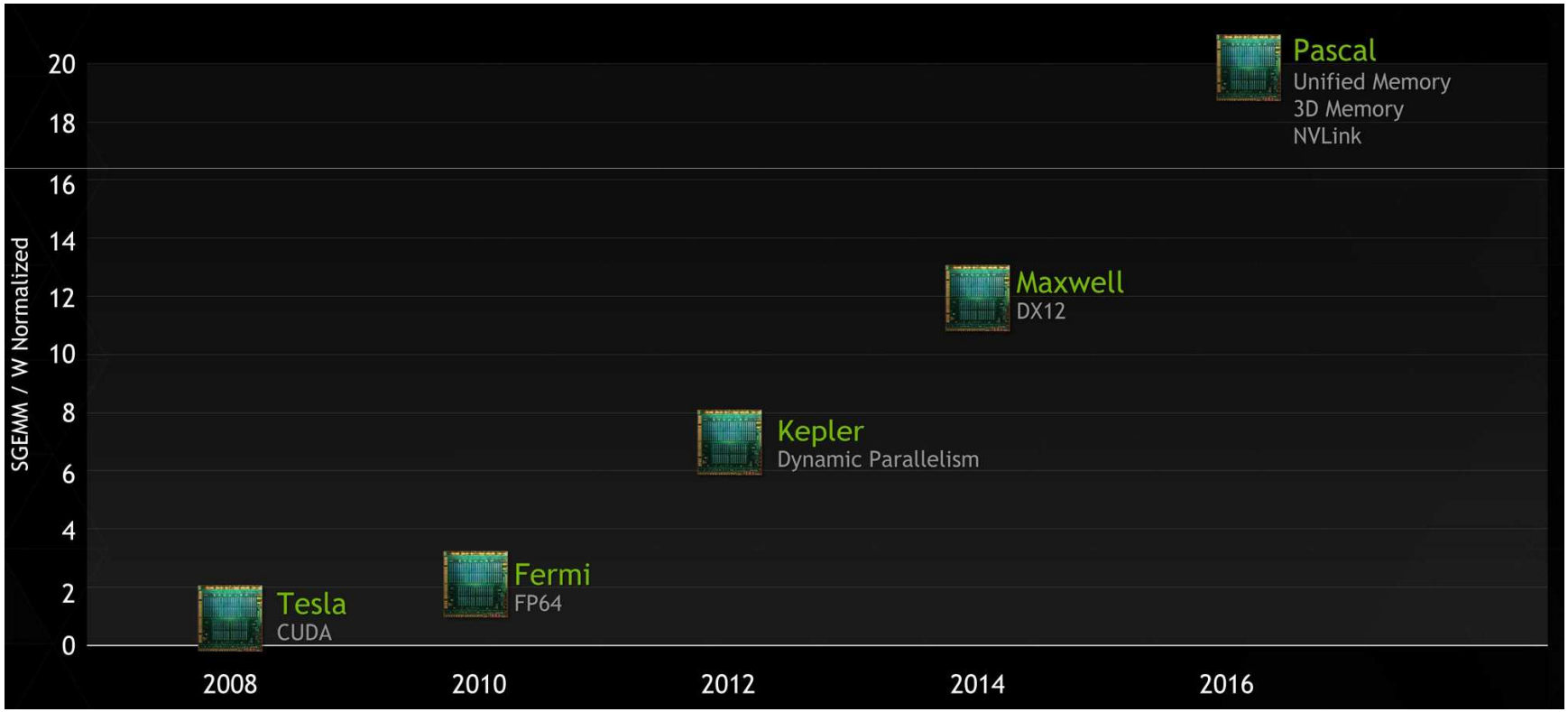
The Maxwell Architecture

- GM 204 (Date: 2015)
  - 5.2 Billion Transistors
  - 2x performance vs. GK104
  - 16 SMM
  - 256-bit GDDR5
  - Maxwell is born for PC Gaming Market (for now)
- Maxwell Streaming Multiprocessor (SMM)
  - 128 CUDA Cores
  - 2x perf/watt vs GK104
  - Improved scheduler
  - New Datapath organization
  - +40% delivered performance per CUDA core



NVIDIA 2016 Roadmap

- What's next?... NVIDIA Pascal architecture (2016)*?*



NVIDIA CUDA Devices

- CUDA-Enabled Devices
- Characterized by:
  - Compute Capability
  - Number of Multiprocessors
  - Number of CUDA Cores

Card	Compute Capability	Number of SMs	Number of SPs
Tegra X1	5.3	2	256
TITAN X	5.2	24	3072
Tesla K80	3.7	2x15	2x2496
GTX TITAN Z	3.5	2x15	2x2880
Tegra K1	3.2	1	192
GTX 690	3.0	2x8	2x1536
GTX 680	3.0	8	1536
GTX 670	2.1	7	1344
GTX 590	2.1	2x16	2x512
GTX 560TI	2.1	8	384
GTX 460	2.1	7	336
GTS 450, GTX 460M	2.1	4	192
GTX 490	2.0	2x15	2x480
GTX 580	2.0	16	512
GTX 570, GTX 480	2.0	15	480
GTX 470	2.0	14	448
GTX 465, GTX 480M	2.0	11	352
GTX 295	1.3	2x30	2x240
GTX 285, GTX 280, GTX 275	1.3	30	240
GTX 260	1.3	24	192
9800 GX2	1.1	2x16	2x128
GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512, GTX 285M, GTX 280M	1.1	16	128
8800 Ultra, 8800 GTX	1.0	16	128
9800 GT, 8800 GT	1.1	14	112

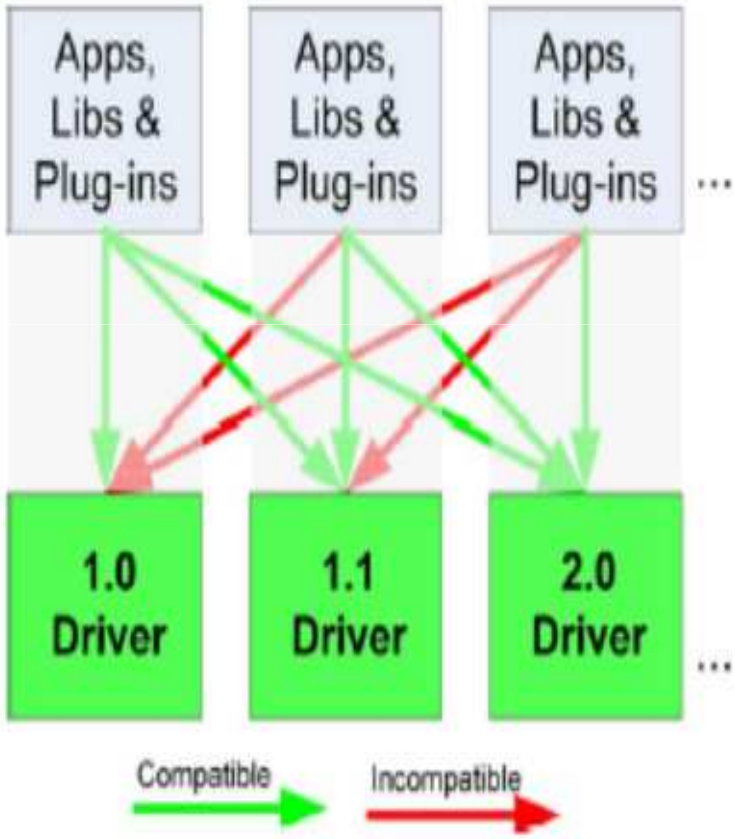
SM: *Stream Multiprocessor (the analog of a CPU core)*
SP: *Stream Processor (the analog of an ALU)*

Device *Compute Capability* vs. CUDA version

- "Compute Capability of a Device" refers to hardware
  - Defined by a major revision number and a minor revision number
  - Example:
    - Tesla C1060 is compute capability 1.3
    - Tesla C2050 is compute capability 2.0
    - Fermi architecture is compute capability 2
    - Kepler architecture is compute capability 3
    - Titan X is compute capability 5.2
  - A higher compute capability indicates a larger set of features available from the hardware
- The "CUDA Version" indicates what version of the software you are using to write code
  - right now, the most recent version of CUDA is 7.5

Compatibility Issues

- The basic rule: the CUDA Driver API is backward, but not forward compatible
  - makes sense: the functionality in later versions increases, and was not there in previous versions

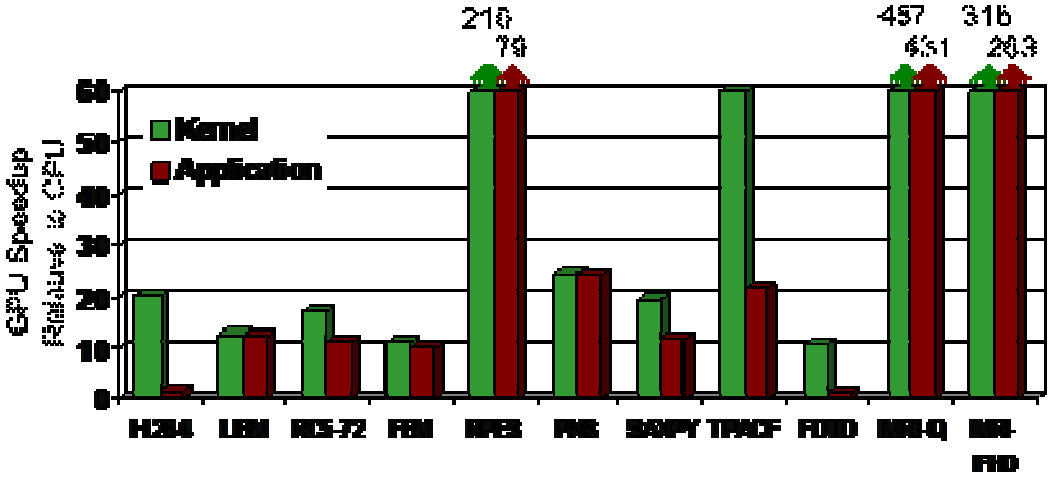


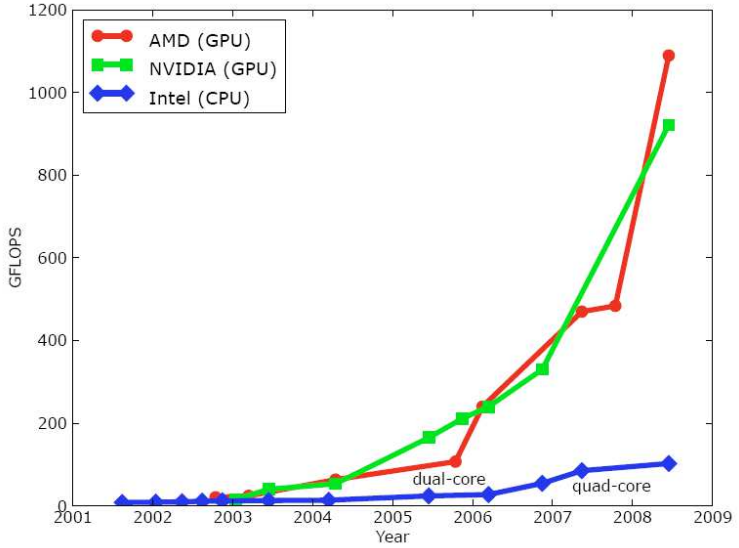
GPUs and current trends in computing

- Many new applications in today's mass computing market have been traditionally considered "supercomputing applications"
  - Molecular dynamics simulation, Video and audio coding and manipulation, 3D imaging and visualization, Consumer game physics, and virtual reality products, …
  - These "super-apps" represent and model physical, concurrent world
- Various granularities of parallelism exist, but…
  - programming model must not hinder parallel implementation
  - data delivery needs careful management
- GPU architectures and related programming models seem to meet particularly well this emerging scenario in computing

GPU performance gains over CPUs

- E.g.: GeForce 8800 GTX vs. 2.2GHz Opteron 248
- Compute power:
  - 500+ GFLOPS vs. 10+ GFLOPS (roughly)
- Memory Bandwidth:
  - 100+ GB/s vs. 10+ GB/s (roughly)
- **10**× speedup in a kernel is typical
  - as long as the kernel can occupy enough parallel threads
- As high as **25**× to **400**× speedup
  - if the function's data requirements and control flow suit the GPU and the application is optimized

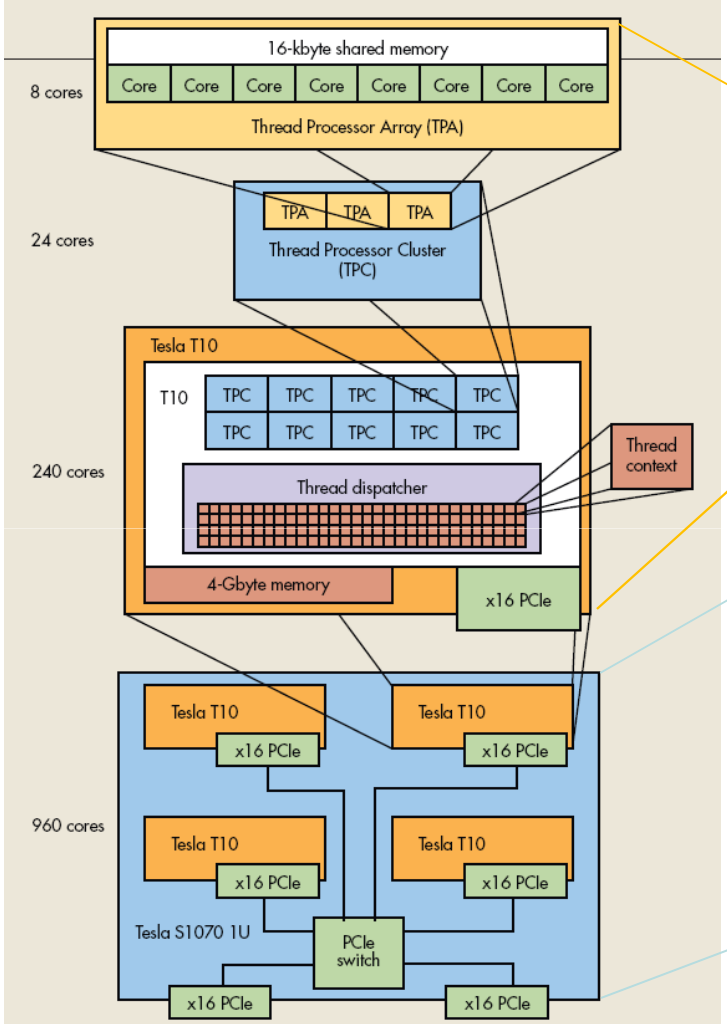




GPUs and HPC: Top500

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
8	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301
9	DOE/NNSA/LLNL United States	**Vulcan** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393,216	4,293.3	5,033.2	1,972
10	Government United States	Cray CS-Storm, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, Nvidia K40 Cray Inc.	72,800	3,577.0	6,131.8	1,499

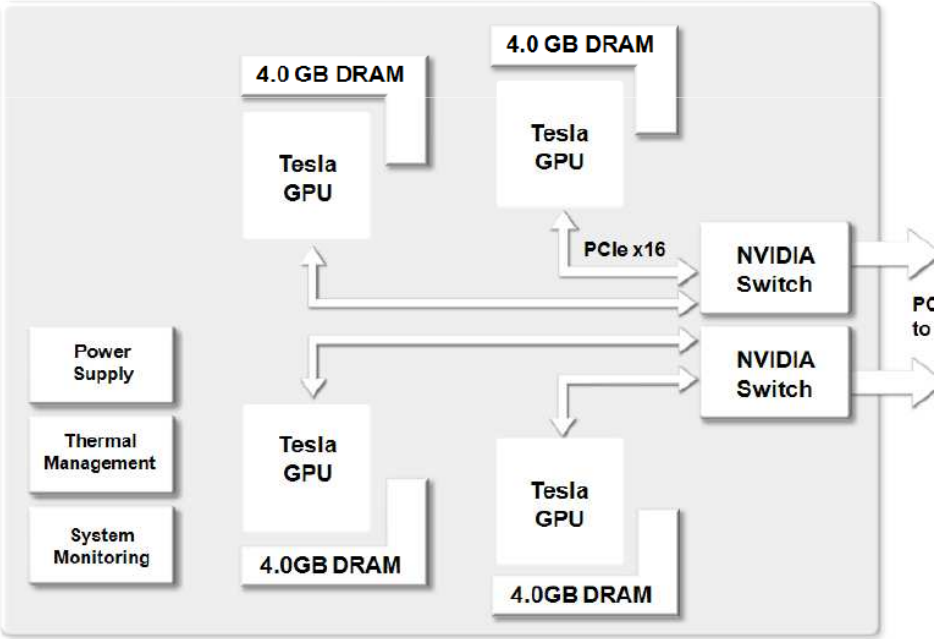
GPUs and HPC: Tesla S1070 blade



1. The Tesla S1070 1U rack-mount system contains 960 cores within four Tesla T10 PCI Express boards. Dual x16 PCI Express connections link the system to a host.

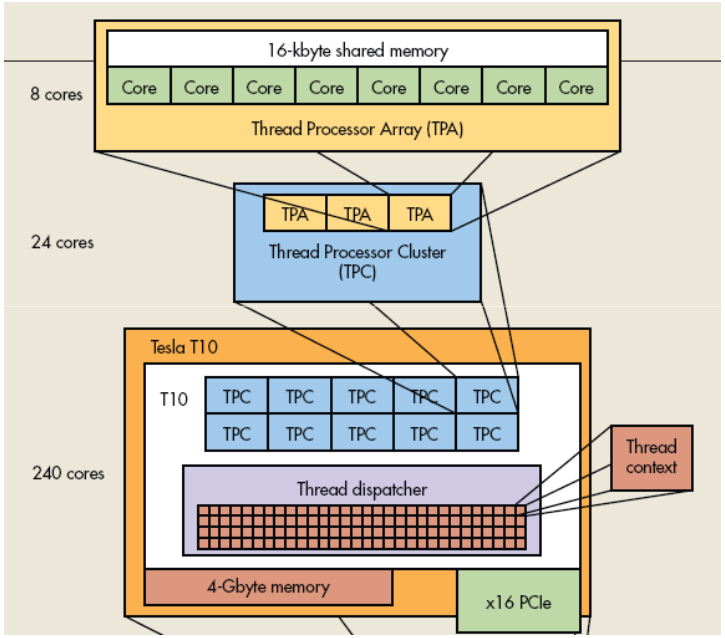
Tesla S1070 blade

- up to 4 teraflops
- 1U rack-mount system
- 240 computing cores per processor
- Frequency of processor cores:
  - 1.296 to 1.44 GHz
- SP FP peak:
  - 3.73 to 4.14 TFlops
- DP FP peak:
  - 311 to 345 GFlops
- Memory Bandwidth:
  - 408 GB/sec
- Dual PCI Express 2.0
- Max Power: 800 W



Tesla T10

- 240 streaming processors/cores (SPs) organized as 30 streaming multiprocessors (SMs) in 10 independent processing units called Thread Processors/Clusters (TPCs)
- A TPC consists of 3 SMs; A SM consists of 8 SPs
- Collection of TPCs is called Streaming Processor Arrays (SPAs)



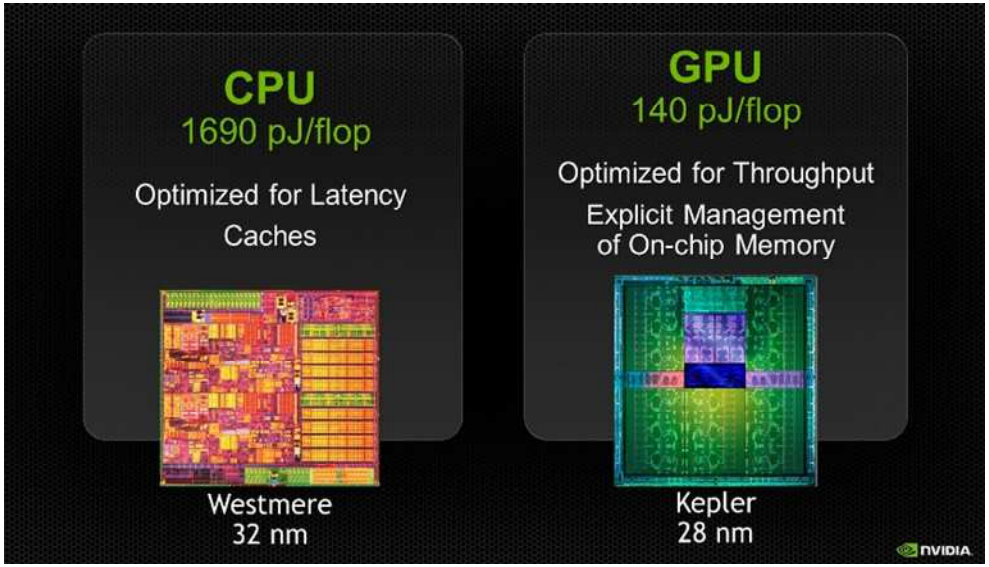
Titan Supercomputer Oak Ridge National Laboratory

- Nov 2012: World's fastest computer (TOP500 list)
- 18,688 NVIDIA Tesla K20X GPUs
- 20 petaflops
- Upgraded from Jaguar supercomputer
  - 10 times faster
  - 5 times more energy efficient than 2.3-petaflops Jaguar system
  - while occupying the same floor space

http://nvidianews.nvidia.com/Releases/NVIDIA-Powers-Titan-World-s-Fastest-Supercomputer-For-Open-Scientific-Research-8a0.aspx#source=pr

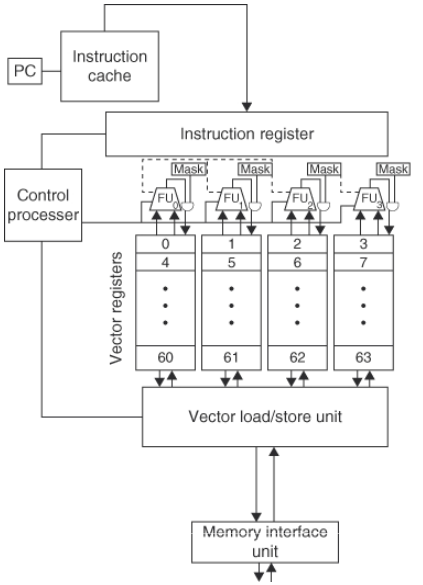
GPU energy efficiency

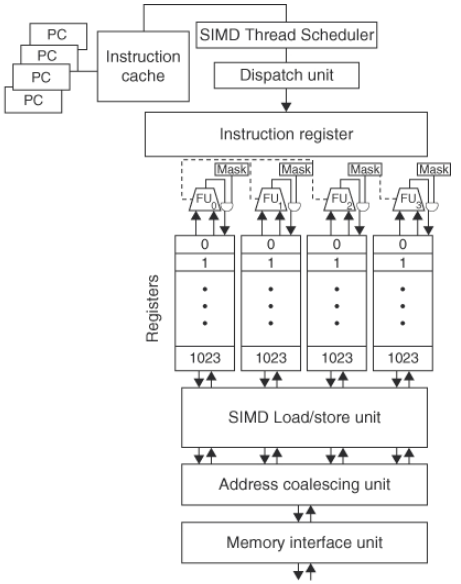
- CPU vs. GPU
- Energy consumed by a single floating point operation:
  - 1690 pJ/flop (CPU) *vs*. 140 pJ/flop (GPU)
- Latency vs. Throughput
- Caches vs. Explicit Managment of On-chip Memory



Vector Processors vs. GPU

- An example of a vector processor with 4 lanes *vs*. a multithreaded GPU processor with 4 SIMD Lanes (bottom figure)
  - GPUs typically have 16 or (many) more SIMD Lanes
- "Control processor" in the Vector system:
  - supplies scalar operands for scalar-vector operations
  - increments addressing for unit and non-unit stride accesses to memory
  - performs other accounting-type operations
- Peak memory performance
  - only occurs in a GPU when the *Address Coalescing* unit can discover localized addressing
  - Similarly, peak computational performance occurs when all internal mask bits are set identically
  - Note: the SIMD Processor has one PC per SIMD thread to help with multithreading





Vector Processors vs. GPU

- GPU *Grid* and *Thread Block* are abstractions for programmers
- “SIMD” Instruction on GPU = Vector instruction on Vector
- Instructions of each thread is 32-element wide
  - thread block with 32 threads =
    strip-minded vector loop with a length of 32 in a vector processor
- Loops in Vector Processors and GPUs:
  - both rely on independent loop iterations
- GPU:
  - Each iteration becomes a thread on the GPU
  - Programmer specifies parallelism
  - grid dimensions and threads/block
  - Hardware handles parallel execution and thread management
  - Trick: have 32 threads/block, create many more threads per multi-processor to hide memory latency

Vector Processors vs. GPU

- Conditional Statements

  Vector:

  – mask register part of the architecture

  – Rely on compiler to manipulate mask register

  GPU:

  – Use hardware to manipulate internal mask registers

  – Mask register not visible to software

  – Both spend time to execute masking

- Gather-Scatter

  GPU:

  – all loads are gathers and stores are scatters

  – Programmer should make sure that all addresses in a gather or scatter are adjacent locations