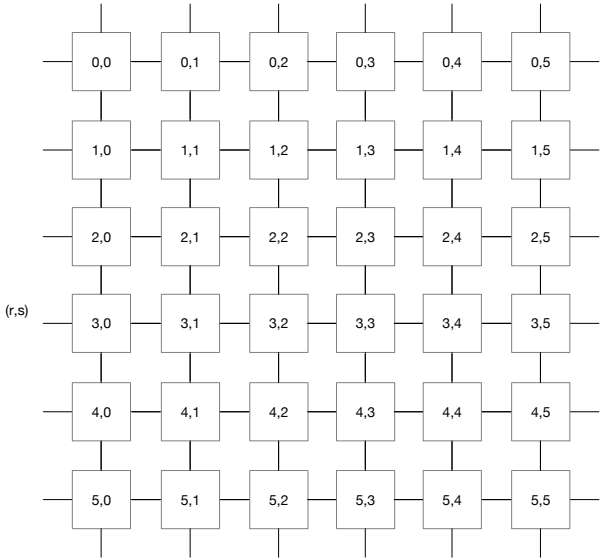
CS420 – Lecture 15

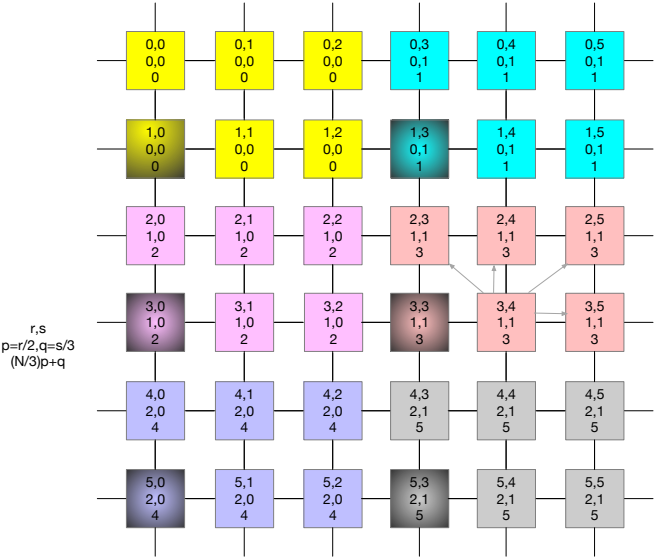
Marc Snir

Fall 2018

MD – process mesh (Cartesian topology)

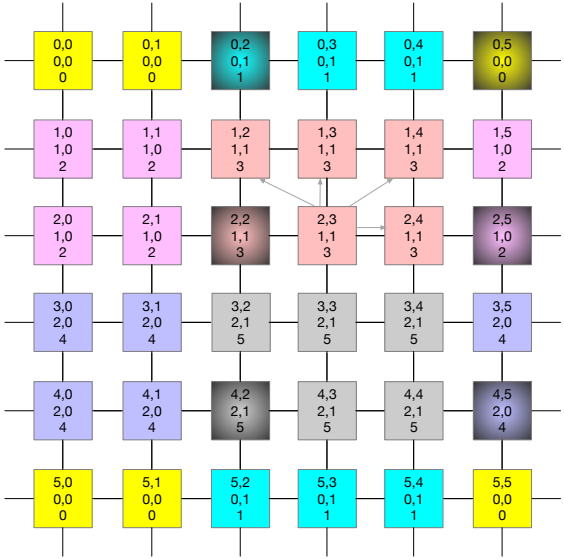


Splitting communicators

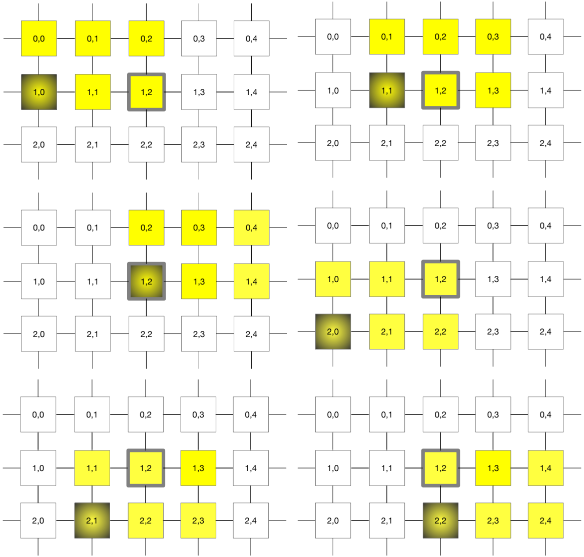


Splitting communicators shift (1,1)

r,s
p=(r+1)/2,q=(s+1)/3
(N/3)p+q



Communicators defined at process (1,2)



- Each process calls `MPI_Comm_split` 6 times (i-shift of 0 and 1 and j-shift of 0,1 and 2).
- In one of the calls it is passes argument `color=MPI_UNDEFINRRED`
- Each process constructs 5 communicators each containing 5 processes
- In each of these communicators, the process is in a different position of a 2×3 rectangle that includes it (except the bottom left position)

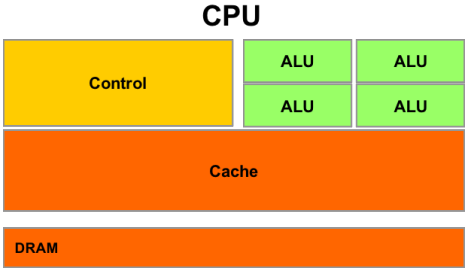
GPGPU
General Purpose Graphic Processing Unit
(with focus on NVIDIA)

GPUs

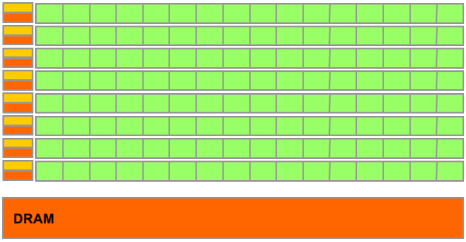
- Started as specialized graphic engines (70s-90s)
  - Leveraged massive parallelism – many pixels and per pixel processing
  - Supported stream processing: a sequence of transformations (kernels) applied to each part of the image
  - Used fast memory (frame buffer)
- Became increasingly programmable, to support higher level graphics
- Started being used outside graphics (GPGPU, 00s)
- GPUs specialized as scientific computing accelerators become available (00s)
  - 64 bit, ECC
- Increasingly focused on deep learning
  - Low precision linear algebra
- Can provide better cost-performance than CPUs for throughput-oriented computations

Different processor designs match different needs

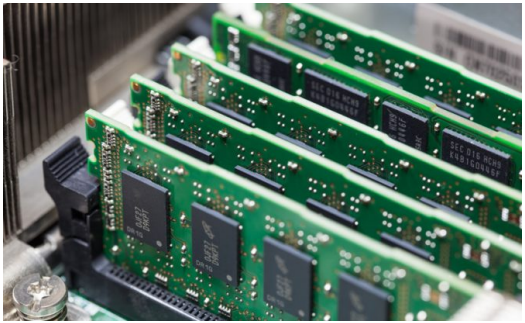
- Conventional CPU: Latency oriented – Optimized for faster execution of sequential code
  - High clock speed
  - Large memory with low latency (DDR3/DDR4)
  - Large caches to further hide memory latency
  - Branch prediction, out of order execution
  - Short ALU pipelines
- Consumes more energy per operation and requires more silicon
- Best match for irregular code with limited parallelism

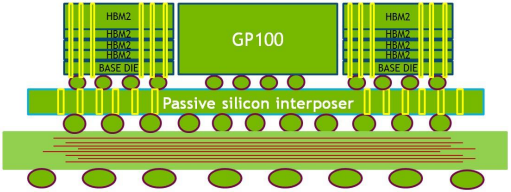


- Graphic Processing Unit (GPU): Throughput oriented
  - Slower clock speed
  - Higher bandwidth, higher latency, lower capacity memory (HBM)
  - small caches to improve memory throughput (aggregation)
  - Simpler control
  - Many ALUs
  - High level of "concurrent multithreading" (multiple streams)
- Consumes less energy per operation and requires less silicon
- Best match for regular code with massive parallelism



DDR vs. HBM




HBM2
HBM2
HBM2
HBM2
BASE DIE
GP100
HBM2
HBM2
HBM2
HBM2
BASE DIE
Passive silicon interposer

Heterogeneous computing

Old idea: Combine mutiple types of compute units to get the best of both worlds

- CPUs can be >10x faster than GPUs on some codes
- GPUs can be >10x times faster than CPUs on some codes
- ⇒ Execute each part of the code on the most suitable engine
  - In particular, execute sequential code on CPU and suitable parallel code on GPU

Questions:

- How are GPUs programmed?
- How is control moved from CPU to GPU and back?
- How is data moved from CPU to GPU and backs.

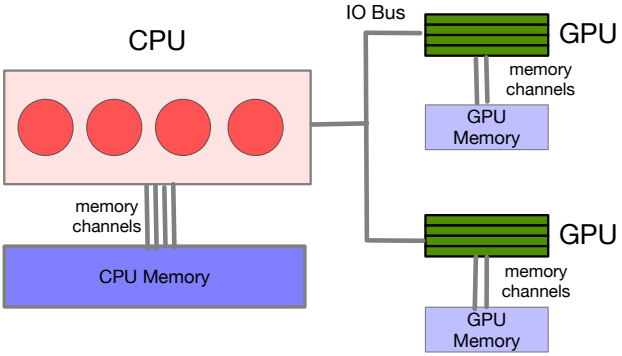
How do we program GPUs?

- OpenCL – standard supported by Apple, AMD, ARM, Intel, based on C++
- CUDA – supported by NVIDIA; available for C, C++ and Fortran.
- OpenACC – OpenMP-like language developed by Cray and NVIDIA
- OpenMP 4.5 – OpenMP extensions designed to handle GPUs
- CUDA is considered easier to use than OpenCL but is proprietary
- OpenACC is usually ahead of OpenMP in GPU support and quality of implementation

How do we program GPUs?

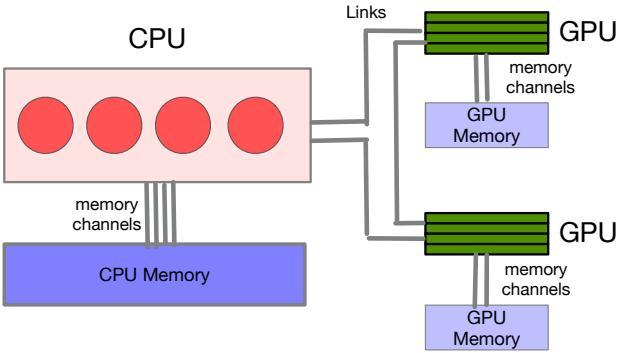
- OpenCL – standard supported by Apple, AMD, ARM, Intel, based on C++
- CUDA – supported by NVIDIA; available for C, C++ and Fortran.
- OpenACC – OpenMP-like language developed by Cray and NVIDIA
- OpenMP 4.5 – OpenMP extensions designed to handle GPUs
- CUDA is considered easier to use than OpenCL but is proprietary
- OpenACC is usually ahead of OpenMP in GPU support and quality of implementation

System with accelerators – current



- CPU memory larger and optimized for low latency
- GPU memory smaller and optimized for high throughput
- CPU and GPU can only access their own memory
- Data can be copied from one memory to the other (DMA)

System with accelerators –unified memory

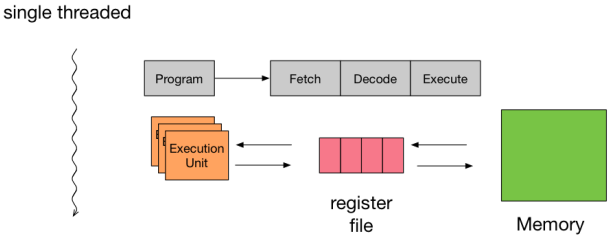


- Each compute engine can access all of the memories
- But performance much better if it accesses its own memory and cache coherence operations are expensive

Execution models – core level

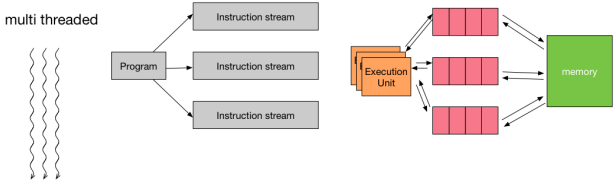
Single-threaded

- One stream of instructions (one instruction counter)
- One set of registers (register file)



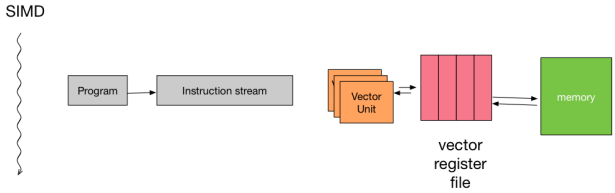
Multi-threaded

- Several streams of instructions
- Several register files
- Instruction streams are independent of each other (but share resources)



SIMD – Single instruction, multiple data

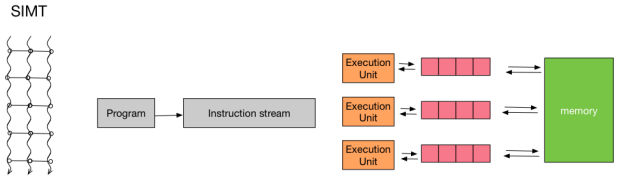
- One stream of instructions
- Vector register file
- Each (vector) instruction specifies the same operation executed on multiple items.
- Can be combined with multithreading



GPU model

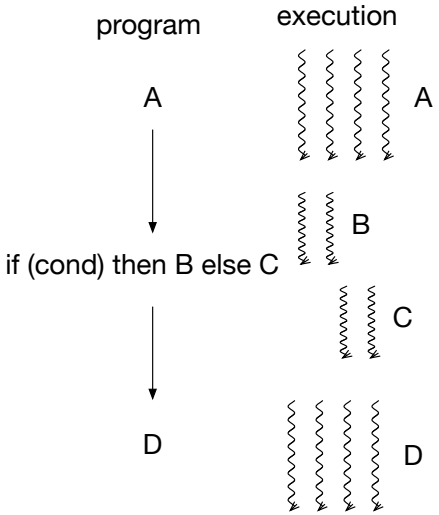
SIMT – Single instruction, multiple threads

- One stream of instructions
- Executed by multiple execution units, each with its own register file



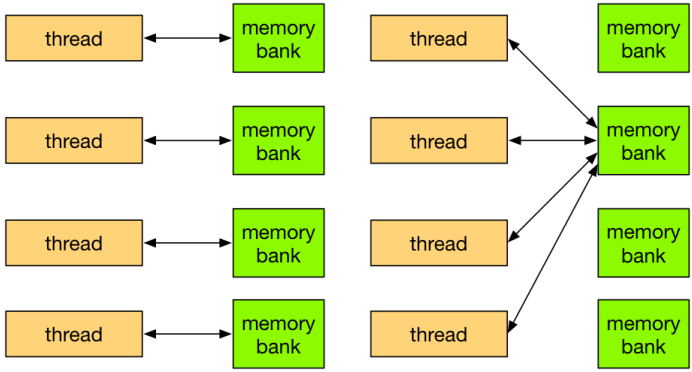
How is SIMT different from SIMD?

- Not much until a branch is encountered: Different threads could branch in different directions
- SIMT execution will continue, with each branch evaluated in turn
- Execution is correct, but performance will suffer



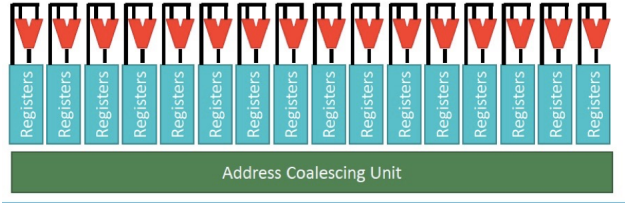
Problem – memory accesses

- Each thread can load from a different address
- Good case: accesses go to different memory banks
  - Contiguous adresses are good (memory is interleaved)
- Bad case: random accesses
  - Conflicts will slow accesses and reduce memory bandwidth



Paliating memory bank conflicts

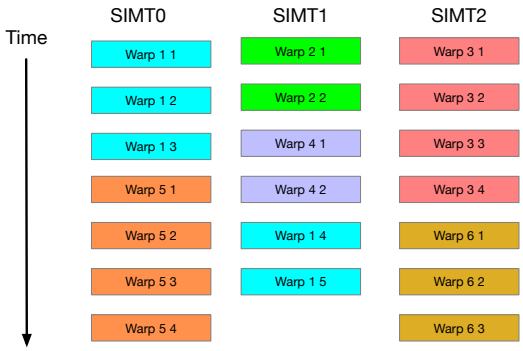
- Address coalescing unit repackages memory accesses so as issue nonconflicting accesses
- Takes advantage of large buffer of pending accesses and (small) caches
- Similar logic exists in DDR memory controllers, but buffer sizes are smaller there



GPU Parallelism in the large

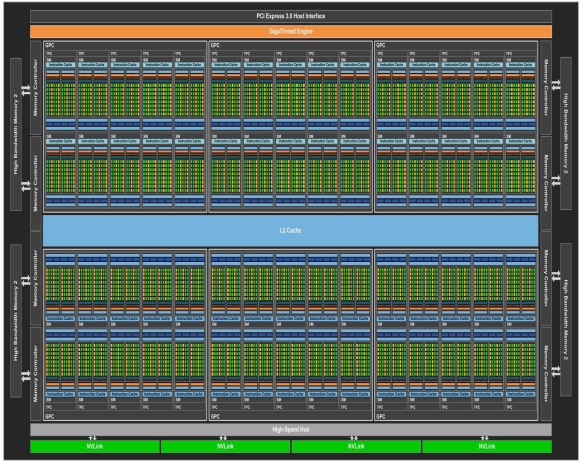
- CPU hides memory latency using multithreading
  - Fixed number of concurrent threads
  - If one thread is waiting for data to arrive from memory, an instruction from another thread can execute

  GPU hides memory latency using multiple *warps* (Nvidia terminology)
  - Warp: bunch of 32 threads executing in SIMT mode
  - If a warp is waiting for data to arrive from memory, it is descheduled and another warp executes.
  - Warp scheduler manages dependencies between warps



Example: Pascal (2016)

GPU = (up to) 60 *streaming multiprocessors* – each SM has 2 blocks



- Each block has 32 cores
- Blocks execute *warps*: a SIMT sequential code with 32 threads.
- Each SM can hold (up to) 64 warps: Two execute, the others wait to be scheduled
  - *Simultaneous multithreading* (CPU): 2-4 threads execute simultaneously; core maintains the state of each.
  - *Concurrent multithreading* (GPU): One warp executes; when it stalls, it is replaced by another one.
- 64K shared L1 per SM (32K per block, 1K per core)
- 4 MB shared L2 per GPU ($\approx$ 1K per core)
  - Caches are use for fast synchronization between cores and for memory access aggregation – not for leveraging temporal locality

Pascal

- 1.328 - 1.480 GHz
- 60×64=3840 32-bit cores
- 120 concurrent instruction streams (one per block)
- 4096-bit memory interface (vs. 64 bits in DDR)
- 255 128-bit registers per core; total of 15,300 KB register file

Typical execution model

CPU thread dispatches *kernel* to GPU, specifying

- Which GPU will execute (if there are several)
- What code will execute
- What data needs to be copied from CPU memory to GPU memory before kernel executes
- What data has to be copied back once kernel has completed

Thread waits for kernel to complete (not necessary)