Contents

Multiplication Table

	2	4	8	16	32	64	128	256	512	1024	2048
2	4	8	16	32	64	128	256	512	1'024	2'048	4'096
4	8	16	32	64	128	256	512	1'024	2'048	4'096	8'192
8	16	32	64	128	256	512	1'024	2'048	4'096	8'192	16'384
16	32	64	128	256	512	1'024	2'048	4'096	8'192	16'384	32'768
32	64	128	256	512	1'024	2'048	4'096	8'192	16'384	32'768	65'536
64	128	256	512	1'024	2'048	4'096	8'192	16'384	32'768	65'536	131'072
128	256	512	1'024	2'048	4'096	8'192	16'384	32'768	65'536	131'072	262'144
256	512	1'024	2'048	4'096	8'192	16'384	32'768	65'536	131'072	262'144	524'288
512	1'024	2'048	4'096	8'192	16'384	32'768	65'536	131'072	262'144	524'288	1'048'576
1024	2'048	4'096	8'192	16'384	32'768	65'536	131'072	262'144	524'288	1'048'576	2'097'152
2048	4'096	8'192	16'384	32'768	65'536	131'072	262'144	524'288	1'048'576	2'097'152	4'194'304

Compilers and Queues

- Version control (e.g. "git"): clone, pull, add, commit, push.
- Phases of a compiler (compile, link).
  - `gcc -O3 -ffast-math -mavx2 -o cpi cpi.c gettime.c`
  - Compile Phase:
    - Lexical Analysis (Scanner): breaking the source code into tokens
    - Syntax Analysis (Parser): analyze the syntactic structure
    - Semantic Analysis: checks the semantic correctness
    - Intermediate Code Generation: intermediate representation
    - Code Optimization: various optimizations on the intermediate code
    - Code Generation: translating the optimized intermediate code into the target machine code
    - Symbol Table Management: compiler maintains a symbol table that keeps track of all identifiers (variables, functions, etc.) used in the program
  - Link Phase:
    - Object Module Generation: compiler produces object files containing the machine code for individual source files
    - Linker: combines multiple object files and libraries into a single executable file
    - Address Binding: assigns final memory addresses to the variables and functions in the program
    - Dynamic Linking (Optional): allowing certain portions of the program to be linked at runtime.
    - Loader: loads the executable file into memory for execution
  - Separate Compilation:
    - each source code file is compiled into a separate object file, which contains the machine code for the functions defined in that file
    - indicated by the `-c` flag
    - Benefits:
      - smaller, more manageable modules
      - you only need to recompile that file and relink the object files

      - Object files can be reused
    - Example:

```
gcc -O3 -ffast-math -mavx2 -c -o cpi.o cpi.c
gcc -O3 -ffast-math -mavx2 -c -o gettime.o gettime.c
gcc -O3 -o cpi cpi.o gettime.o
```

- Importance of compiler optimization and how can affect the results.
  - i.e. it won't automatically optimize
  - `-O` gives the same results as `-O1`
  - `-ffast-math`: Reordering of operations, Allowing reassociation, Ignoring NaN, Relaxing precision requirements
  - `-mavx2`: instructs the compiler to enable support for the Advanced Vector Extensions 2 (AVX2) instruction set
- Basics of "make":

```
CFLAGS=-O3 -ffast-math -mavx2
CC=gcc

cpi : cpi.o gettime.o

cpi.o : cpi.c gettime.h

gettime.o: gettime.c gettime.h

clean:
  rm -f cpi cpi.o gettime.o
```

Then:

```
make
gcc -O3 -ffast-math -mavx2 -c -o cpi.o cpi.c
gcc -O3 -ffast-math -mavx2 -c -o gettime.o gettime.c
gcc -o cpi cpi.o gettime.o
```

- Purpose and function of batch queue systems (e.g. SLURM). Scheduling: priority versus FIFO. How you request resources (nodes, cores, etc.)
  - SLURM:
    - Simple Linux Utility for Resource Management
    - Manages resources like compute nodes, processors (cores), memory, and other hardware components in a cluster.
    - Prevents resource conflicts by allocating exclusive access to resources for individual jobs.
  - Fifo: first in first out
  - Lifo: last in first out
  - Batch system: mostly Fifo, but has priority slots

BASH

- Redirection: `ls *.c > output.dat`
  - used for redirecting output to a file or overwriting a file
  - `>>` is used to append
- Piping: `ls | wc`
  - take the output of one command and feed it as input to another command
- `wc`: Lines, Words, Characters
- Variables:

```
TEST = 12;
echo $TEST1; 12
echo "$TEST1"; 12
echo '$TEST1'; $TEST1
```

- Executable Scripts:

```
cat script.sh; #!/bin/bash;
chmod +x script.sh
```

- “HERE” Documents:

```
cat script;
#!/bin/bash
A="the variable A"
CAT <<EOF
Even more impressive is
that we can have variables.
Parameter 1: $1
Variable A: $A
EOF

./script "parameter 1"
Even more impressive is
that we can have variables.
Parameter 1: parameter 1
Variable A: the Variable A
```

- Return Code: `$?`
  - `0`: Success
  - `1`: General error (i.e. `false`)
  - `2`: Misuse of shell built-ins (syntax or wrong commands, e.g. “No such file”)
- Test:
  - `if test $A -gt 10; then echo large; fi` - a numerical comparison operator
  - `if [ $A -gt 10 ]; then echo large; fi` - conditional expressions
  - `if [[ $A -gt 10 ]]; then echo large; fi` - string comparison
- For Loop:
  - `for V in a b c; do echo $V; done`
  - `for (( i=0; i<10; ++i)); do echo $i; done`
- Shift:

```
while test -n "$1"; do #while not empty
   echo "Processing $1"
   shift
```

- Arithmetic:
  - `A=90; B=$((A/2+7)); echo B; 52`
  - `ANSWER = $(echo "scale=10;1.0/3.0" | bc); echo ANSWER; 0.3333333333`
- GREP:
  - `-c`: count
  - `-q`: quiet
  - `-i`: case insensitive search
  - `-A 1`: number of lines to display after each matching line

  - `-B 1`: number of lines to display before each matching line
  - `-C 1`: number of lines to display both before and after each matching line
  - greedy match: `egrep 'all.*s' interim.txt`
  - minimum match: `egrep 'all.*?s' interim.txt`
- REGEX Capture:

```
cat retest.sh
#!/bin/bash
RE="Phase ([0-9]) complete, ([0-9]+\.[0-9]+) seconds"
while read A; do
  if [[ "$A" =~ $RE]]; then
    echo "Match: ${BASH_REMATCH[0]}" #print whole line
    echo "Time: ${BASH_REMATCH[2]}" #print second regex match in line
  fi
done

./retest.sh < report.txt
Match: Phase 1 complete, 3.11 seconds
Time: 3.11
Match: Phase 1 complete, 10.92 seconds
Time: 10.92
Match: Phase 1 complete, 1.15 seconds
Time: 1.15
```

- Short Circuits:

```
Roman@MINGW64 ~/Desktop (main)
$ if echo first || echo second; then echo yes; fi
first
yes

Roman@MINGW64 ~/Desktop (main)
$ if echo first || echo second || echo third; then echo yes; fi
first
yes

Roman@MINGW64 ~/Desktop (main)
$ if echo first || echo second  && echo third; then echo yes; fi
first
third
yes

Roman@MINGW64 ~/Desktop (main)
$ if false || echo second  && echo third; then echo yes; fi
second
third
yes

Roman@MINGW64 ~/Desktop (main)
$  if ! echo first || echo second; then echo yes; fi
first
second
yes
```

Performance

- Moore's Law and how it relates to the Top500 list
  - "The number of transistors that can be placed on an integrated circuit at a reasonable cost doubles every two years"
  - Top500 is only about performance, but it still follows the pattern.
- Strong and weak scaling
  - Strong Scaling: focuses on fixing the problem size and increasing the number of processors to improve performance
    - Amdahl's Law: "The theoretical maximum speedup obtained by parallelizing a code ideally, for a given problem with a fixed size"
    - $\text{Speedup}(N) = \frac{T_S}{T_p(N)} = \frac{T_S}{\alpha T_S + (1-\alpha)\frac{T_S}{N}} = \frac{1}{\alpha + \frac{1-\alpha}{N}}$
    - $\text{Speedup}(N) \to \frac{1}{\alpha}$ as $N \to +\infty$
      - T_S: execution time of the serial code
      - T_p: execution time of the parallel code
      - α: fraction of the code that is not parallel
      - N: number of processors
  - Weak Scaling: increasing both the problem size and the number of processors in proportion to maintain a consistent workload per processor.
    - Gustafson's Law: "The theoretical maximum speedup obtained by parallelizing a code ideally for a problem of constant size per core"
    - $\text{Speedup}(N) = \frac{T_S}{T_p(N)} = \alpha + (1-\alpha)N$
    - $\text{Speedup}(N) \to (1-\alpha)N$ as $N \to +\infty$
  - Weak is easier to show
- Latency and bandwidth.
  - Latency: time
  - Bandwith: data per time
- Memory, cache and bus (e.g., attached network or GPU card) hierarchy and their relative performance.
  - Memory:
    - Registers: few nanoseconds
    - Cache Memory
    - Main Memory (RAM): tens of nanoseconds
    - Secondary Storage (e.g., Hard Drives, SSDs): milliseconds or more
  - Bus:
    - Front-Side Bus (FSB): Connects the CPU to the memory
    - Memory Bus: Connects the CPU to the main memory (RAM)
    - System Bus (e.g., PCI Express): Connects various components like GPUs, network cards, and other peripherals to the CPU and memory.
- Different ways of instrumenting (benchmarking) your code.
  - time:
    - `time ./body`
    - `real 0m3.976s`: The actual elapsed time from start to finish, including waiting for external resources, I/O operations, and any other delays.
    - `user 0m3.9283`: The CPU time spent in user-mode code while executing the actual instructions in the user-level code of the program.
    - `sys 0m0.037s`: The CPU time spent in the kernel where it executes system calls and other kernel-level operations on behalf of the program.
  - `-g` flag:
    - tell `gcc` to emit extra information for use by a debugger
    - Allows for easier debugging with tools like `gdb` (GNU Debugger)
    - Increases the size of the compiled binary.
  - `-Og` flag:
    - Enables optimizations that do not interfere significantly with debugging
  - `perftools-lite`:
    - must use a queue system
    - Table 1: profile by Function
    - Table 2: functions, and line numbers within functions, that have significant exclusive

        sample hits, averaged across ranks.

  - Self timing:
    - Get the current time
    - Do some calculations
    - Get the new time
      - `time()`: uses seconds
      - `gettimeofday()`: uses seconds and microseconds
      - `clock_getres()`: finds the resolution/precision of the clock
      - High resolution C++ timer: `std::chrono: high_resolution_clock`, uses smallest tick period provided by the implementation.
      - Cycle Counters: `#include "cycle.h"; ticks getticks(void); double elapsed(ticks t1, ticks t0);` returns a double-precision variable in arbitrary units for comparison of time intervals.

OpenMP

- OpenMP uses a shared memory paradigm
  - OpenMP is a multithread model, within a single process. Communications between threads are implicit. The management of communications is under the responsibility of the compiler (and the operating system).
  - Data are shared implicitly within the node through the Random Access Memory.
  - MPI uses a distributed memory paradigm: Data are transferred explicitly between nodes through the network
- An Open MP program is executed by only one process, called the Master thread
- The master thread activates light-weight processes, called the workers or slave threads at the entry of a parallel region
- Basics of OMP "make":

```
CFLAGS=-O3 -ffast-math -mavx2 -fopenmp
LDFLAGS=-fopenmp #Linker flags
CC=gcc
cpi :
  cpi.o gettime.o
cpi.o :
  cpi.c gettime.h
gettime.o :
  gettime.c gettime.h
clean:
  rm -f pi cpi.o gettime.o
```

- Directive based: `#pragma omp directive-name [clause[ [,] clause] ] new-line`
- Model (shared memory, threads, a "thread" runs on a "core")
- The serial and "parallel" (region) parts.
- The "parallel for" loop:
  - Loops without loop indices or while loop are not supported by OpenMP
  - The end of a parallel for is always a barrier - `nowait` directive not possible, because you are leaving a parallel section at the end
- Synchronization between threads, for example "reduce" clause, or "atomic" or "critical" pragmas. Performance of each.
  - reduce:
    - Each thread computes its partial result, which is then combined to the others at the end of the parallel loop.
    - can be carried out in parallel using parallel reduction algorithms
    - `#pragma omp for reduction(+:s) reduction(*:p,r)`
    - Arithmetic: `+, *, -`
    - Logical/Boolean: `&, |, ^, &&, ||`
      - `&`: bitwise and
      - `&&`: logical and
      - `^`: bitwise xor
    - Predefined: `max, min`
    - User defined
  - atomic:
    - forces the shared variable access or update to be performed by one thread at a time. Uses Hardware support.
    - `#pragma omp atomic`
  - critical:
    - Order of execution is non-deterministic, but one thread at a time.
    - Less efficient than atomic for simple operations as it uses locking instead of hardware support.
    - `#pragma omp critical`
- Shared versus private variables:
  - Default: shared
    - `#pragma omp parallel default(none) firstprivate(a)`
  - Some variables are always private:

        - Loop indices are always private integer variables
        - Local variables inside subroutines
        - Variables declared in a parallel region
- How OpenMP schedules work between threads:
    - We don’t say it how to split the loop, but it mostly does this by itself
    - `schedule(static,128)`
        - chunks of size N
        - `#pragma omp for schedule(static,128)`
    - `schedule(runtime)`
        - strategy is set at runtime using the environment variable `OMP_SCHEDULE`
        - `export OMP_SCHEDULE="guided, 16"`
        - `#pragma omp for schedule(runtime)`
    - `schedule(dynamic,128)`
        - chunks of size N, but they are assigned whenever a thread is available
    - `schedule(guided,128)`
        - chunks of exponentially decreasing size, but larger than N
- Memory Issues:
    - Memory access can be the dominate cost
    - Conflicts between threads can lead to poor cache memory management (the so-called cache misses)
    - Level 1 and 2 cache memory management is key to OpenMP performance
    - This can be achieved using a “First Touch” approach

MPI

- MPI uses a distributed memory paradigm
  - MPI is a multi-process model, for which communications between processes are explicit and under the responsibility of the programmer.
  - Data are transferred explicitly between nodes through the network
  - OpenMP uses a shared memory paradigm: Data are shared implicitly within the node through the Random Access Memory.
- MPI Structure:

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]){
  int rank, size;
  MPI_Init(&argc,&argv);                /* Connect processes to each other */
  MPI_Comm_size(MPI_COMM_WORLD,&size);  /* Get total number of processes */
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);  /* Rank of this process */
  ...
  MPI_Finalize();
}
```

- Model (distributed memory, message passing, a "rank" is a process).
  - Threads don't see eachother's memory
- How to split work between "ranks" (also known as load balancing or domain decomposition).
  - Has to be done manually
- How to compile MPI programs (e.g., mpicc, cc, mpicxx, CC, etc.).
  - `mpicc --version: gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0`
  - `mpicxx --version: g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0`
  - `mpif90 --version: GNU Fortran (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0`
  - Cray Wrapper:
    - `cc --version: Cray clang version 11.0.0`
    - `CC --version: Cray clang version 11.0.0`
    - `ftn --version: Cray Fortan Version 11.0.0`
  - `module swap PrgEnv-cray PrgEnv-gnu`
    - `cc --version: gcc (GCC) 10.1. 20200507`
    - `CC --version: g++ (GCC) 10.1.0 20200507`
    - `ftn --version: GNU Fortran (GCC) 10.1.0 20200507`
  - `module swap PrgEnv-gnu PrgEnv-pgi`
    - `cc --version: pgcc (aka pqcc18) 20.1-0 LLVM`
    - `CC --version: pgc++ 20.1-0 LLVM`
    - `ftn --version: pgf90 20.1-0 LLVM`
- Some of the common functions (those covered in the lectures).
  - Send: Performs a blocking send
    - The execution remains blocked until the message is fully received and stored in the target variable
    - `MPI_Send(value, length, mpi_datatype, target_rank, tag, mpi_communicator)`
  - Receive: Performs a blocking receive
    - `MPI_Recv(value, length, mpi_datatype, source_rank, tag, mpi_communicator, status)`
  - Reduce: Reduces values on all processes to a single value
    - `MPI_Reduce(value_source, value_target, length, mpi_datatype, mpi_operation, target_rank, mpi_communicator)`
  - All-Reduce: Combines values from all processes and distributes the result back to all processes
    - `MPI_Allreduce(value_source, value_target, length, mpi_datatype, mpi_operation, mpi_communicator)`
  - Broadcast: Broadcasts a message from the process with rank "root" to all other processes of the communicator
    - `MPI_Bcast(value_source, length, mpi_datatype, source_rank, mpi_communicator)`
  - Scatter: Sends data from one process to all other processes in a communicator.
    - Chunk #i is always sent to processor of rank i.

        - block_length is the same for the source and target arguments in most cases
        - The combination of block_length and mpi_datatype for the send must represent the same amount of data than the combination block_length and mpi_datatype for the receive
        - MPI_Scatter(source_list, block_length, mpi_datatype, target_list, block_length, mpi_datatype, source_rank, mpi_communicator)
    - Gather: Gathers together values from a group of processes
        - Data is collected in process root in the same order as the other processors' ranks.
        - block_length is the same for the source and target arguments in most cases
        - The combination of block_length and mpi_datatype for the send must represent the same amount of data than the combination block_length and mpi_datatype for the receive
        - MPI_Gather(source_list, block_length, target_list, block_length, mpi_datatype, target_rank, mpi_communicator)
    - All-gather: Gathers data from all tasks and distribute the combined data to all tasks
        - block_length is the same for the source and target arguments
        - MPI_Gather(source_list, block_length, mpi_datatype, target_list,block_length, mpi_datatype, mpi_communicator)
    - All-to-All: Sends data from all to all processes (matrix transpose)
        - block_length is the same for the source and target arguments
        - MPI_Alltoall(source_list, block_length, mpi_datatype, target_list, block_length, mpi_datatype, mpi_communicator)
- How message passing works and common problems (e.g., deadlocks).
    - Non-blocking communication helps prevent deadlocks:
        - Send and receive: MPI_Isend() and MPI_Irecv()
            - MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
            - MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
        - Waiting until completion: MPI_Wait()
            - MPI_Wait(MPI_Request *request, MPI_Status *status)
        - Testing if completion: MPI_Test()
            - MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
    - No risk of deadlock but risk of memory leak if the communication is not properly terminated
- User defined MPI Datatypes:

```
struct particle{
  char category[5];
  int mass;
  float coords[3];
  bool valid;
}

//number of elements for each MPI datatype
int datacount[] = {5,1,3,1}

//the offset of each member in the particle structure
MPI_Aint dataoffset[] = {offsetof(particle, category),
offsetof(particle, mass), offsetof(particle, coords),
offsetof(particle, valid)
};

//array of MPI datatypes
MPI_Datatype datatypes[] = {MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_C_BOOL};

//declares a variable named particle_type of type MPI_Datatype
MPI_Datatype particle_type

MPI_Type_create_struct(4, datacount, dataoffset, datatypes, &particle_type);
```

```
/*
This line creates a custom MPI datatype using MPI_Type_create_struct.
It takes four parameters:
  - The number of elements in the datatypes array (4 in this case).
  - The array datacount specifying the number of elements for each MPI datatype.
  - The array dataoffset specifying the offset of each member in the structure.
  - The array datatypes specifying the MPI datatypes of the members.
*/

//particle_type can be used in MPI communication routines
MPI_Type_commit(&particle_type);

...

// frees the resources associated with the custom MPI datatype
MPI_Type_free(&particle_type);
MPI_Finalize();
```

Cloud & Containers

- Difference between Cloud (Virtual Machines) and containers
  - VM: a "virtual" computer. Has memory, CPUs, network, disks, etc.
  - VMs provide strong isolation because each VM runs its own operating system (OS)
  - Containers have lower overhead compared to VMs because they share the host OS and do not require a full OS for each container
- How to handle "persistence" with VMs and containers, e.g., "snapshots".
  - VM snapshots capture the entire state of a VM at a specific point in time, including the disk, memory, and configuration
  - Containers use volumes to handle persistence by storing data outside the container filesystem
- Ephemeral computing (create a resource, compute, throw away the resource).
  - Virtual machines or containers, are provisioned and de-provisioned dynamically as needed, typically for short-lived and temporary workloads.
  - They do not retain persistent data or state between instances
- You are "root" in virtual machines and containers and what this means.
  - In a virtual machine, being "root" means having administrator access to the entire virtualized operating system
  - The root user has full control over system resources, network configurations, and user accounts within the virtual machine
- Dockerhub: like github but for containers
- How to access data
  - images and snapshots for VMs
    - In VMs, disk images and snapshots are typically managed through the hypervisor or virtualization platform
  - mounting host directories for containers.
    - `docker run -it --run --mount type=bind,source=/Users/dpotter,target=/home ubuntu:20.04 /bin/bash`
  - Also, we can create volumes for a VM, like plugging in a USB stick and that can be swapped to another VM.
- Example Docker File:

```
FROM ubuntu:trusty

RUN apt-get update -q && apt-get install -y -q --no-install-recomennds cowsay

RUN ls -s /usr/games/cowsay /usr/bin

COPY . /test-cowsay
WORKDIR /test-cowsay
CMD ./test.sh
```

MapReduce

- Hadoop:
  - HDFS stores large files by dividing them into blocks
  - These blocks are then distributed across multiple nodes in a Hadoop cluster
  - Designed to scale horizontally by adding more commodity hardware to the cluster.
- Raid Levels:
  - Level 0 - striping
    - Maximizes Storage
    - No File recovery
  - Level 1 - mirror
    - Half the Storage
    - Data Redundancy
  - Level 5 - distributed partity
    - Mix of performance, reliability, and cost
    - Provides 66% of the storage, where the third bit of any triplet can be recovered by a calculation over modulo 2, which is stored in a fourth bit, the parity bit.
- Why the "compute" is sent to the "data" instead of the normal "HPC" way.
  - The idea of bringing computation tasks closer to the data they need to process, rather than moving large volumes of data to where the computation is happening
  - Data locality refers to the principle of keeping the data physically close to the computation resources to minimize data transfer and enhance performance
  - When a computation task is initiated, it is scheduled to run on a node that holds a copy of the data it needs
  - Advantages:
    - Minimized Data Transfer
    - Increased Performance
- Data model: write once and read (process) multiple times, As is Google's business model
- What the "map" and "reduce" phases do.
  - `(for F in File?.txt; do cat $F | ./mapper.py; done) | sort | ./reducer.py > DIRECT.txt`
  - Map phase:
    - Read each line from input
    - Remove leading/trailing spaces
    - Remove punctuation
    - Convert to lowercase
    - Split into individual words
    - Output words with count of 1
      - transform input data into a set of key-value pairs
      - User-defined Map function is applied to each input record independently
      - The intermediate key-value pairs are sorted based on their keys to facilitate efficient processing during the Reduce phase
  - Reduce Phase:
    - Input has been sorted by key
    - Read each word and count
    - For repeated words sum counts
      - To process and aggregate the intermediate key-value pairs generated by the Map phase
      - The user-defined Reduce function is applied to each group of values that share the same key.
- What are the "key" and "value"?
  - The key represents a category or grouping, and the value is the associated data
  - The value is the result of the aggregation or processing.

Hybrid Computing

- CPU “sockets” and “cores”, and GPU “SMs” and “cores”.
  - CPU Sockets: physical connector on the motherboard
  - CPU Cores: an individual processing unit within a CPU
  - GPU SMs (Streaming Multiprocessor): consists of multiple CUDA cores (processing units) and other components necessary for parallel processing
  - GPU Cores: a processing unit within an SM on an NVIDIA GPU. It is similar in concept to a CPU core but optimized for parallel processing tasks
  - Eiger: 2 sockets with 64 cores each. GPU has one chip, but SM’s are independent units, which each have cores.
- SIMD (Single Instruction, Multiple Data) (AVX) on CPUs and “Warps” on GPUs.

Table 1: Vectorized instructions on the CPU can follow different standards. AVX would allow up to 4 double precision floats to be processed in parallel.

Standard	Size	Single (32 Bits, 4 Bytes)	Double (64 Bits, 8 Bytes)
AVX	256 Bit, 32 Bytes	8	4
AVX 512	512 Bit, 64 Bytes	16	8

  - SIMD (AVX):
    - SIMD is a parallel processing technique in which a single instruction is applied to multiple data elements simultaneously
    - AVX enables processors to operate on larger vectors of data in parallel
    - AVX supports wider vector registers, allowing operations on more data elements simultaneously. For example, AVX introduces 256-bit and 512-bit vector registers
    - With AVX, a single instruction might perform an operation like adding two vectors of eight single-precision floating-point numbers in a single cycle.
  - Warps:
    - A warp is the basic unit of execution on a GPU, and all threads within a warp execute the same instruction at the same time
    - The warp size is the number of threads in a warp. Common warp sizes are 32 or 64, depending on the GPU architecture
    - Within a warp, all threads execute the same instruction. If a certain condition causes divergence in the code (some threads take a different execution path), the GPU will serialize the execution of divergent threads
- CPU versus GPU
  - CPU: small number of high performance cores and ~ one thead per core.
  - GPU: large number of lower performance cores and many threads per core.
  - Memory bandwidth to memory on each:
    - GPU have a higher bandwith to memory
    - CPU has a memory width of 8 floats
    - GPU has 4x the width, so it gets more throughput
- Divergence: how it is handled on the CPU (SIMD) and on the GPU (Warps).
  - CPU: will not do AVX, it cannot block the instructions. Hard to deal with Divergence here.
  - GPU: if first, else part afterwards, lots of blocking. Takes longer on a GPU.
- Data alignment: what it is and why is it important: If not aligned, more reads than necessary
- Latency hiding and Occupancy on GPUs. Latency to start kernels or data transfer.
- Latency of instruction on the GPU versus the CPU.

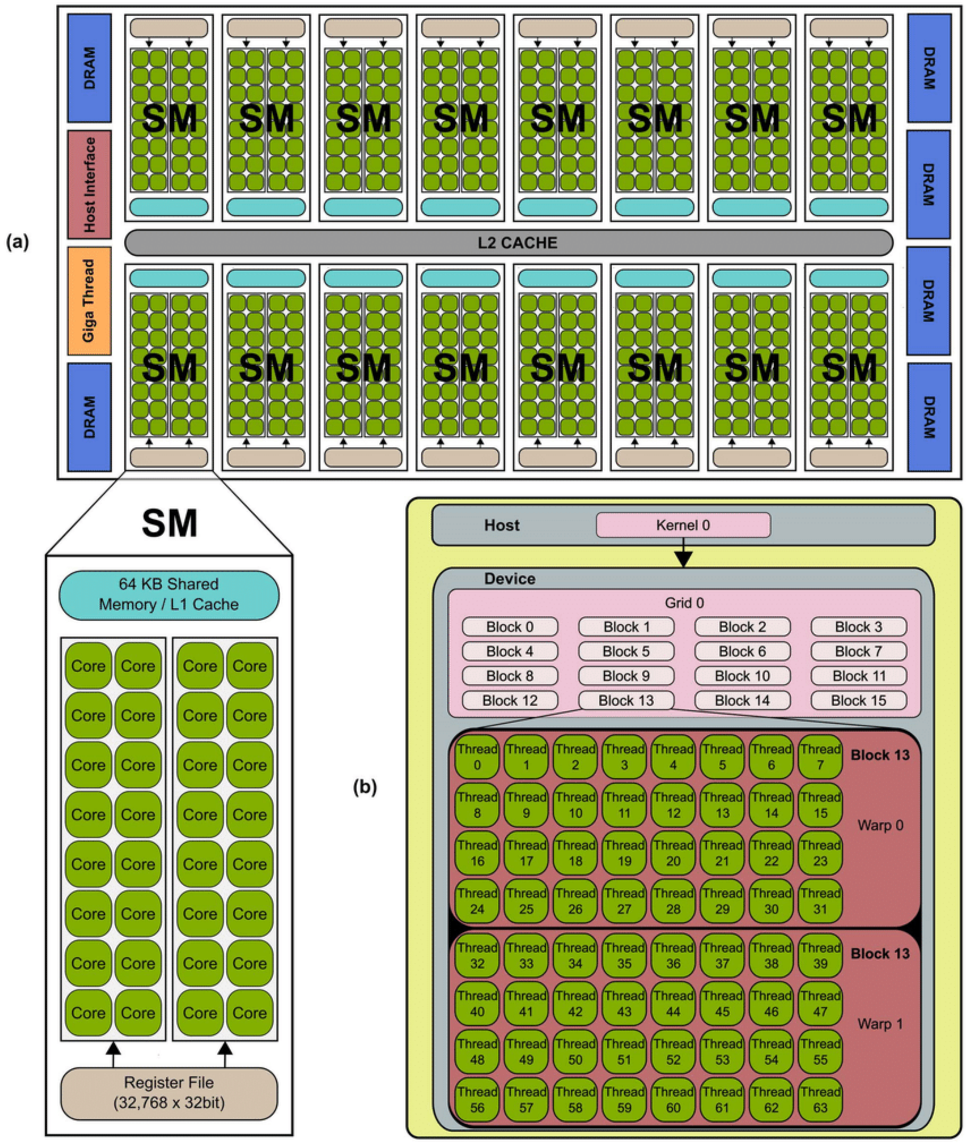


Figure 1: (a) Each SM is comprised of several Stream Processor (SP) cores, as shown for the NVIDIA's Fermi architecture. (b) The GPU resources are controlled by the programmer through the CUDA programming model.

OpenACC

- Directive Based: Disabled if unsupported or if “-acc” not specified

  ```
  module load daint-gpu
  module load cudatoolkit
  module swap PrgEnv-cray PrgEnv-nvidia
  CC --version: nvc++ 21.3-0 LLVM
  CC -O3 -acc -Minfo=acc -o saxpy saxpy.cxx
  ```

- Basic GPU operations: allocate, copy, kernel launch.
  - copy - Allocate and copy variable to the device and copy it back at the end
  - copyin - Allocate and copy to device
  - copyout - Allocate space but do not initialize. Copy to host at the end
    - `#pragma acc data pcreate(x[0:N]) pcopyout(y[:N])`
    - `y[:N]`: length is N
    - `y[:N]` is equivalent to `y[0:N]`
  - create - allocate space but do not initialize or copy back to the host
  - present - the variable is already present on the device (when data regions are nested):
    - `copy`, `copyin`, `copyout`, and `create`: create and copy occur, but there is an error if the data is already present
    - `pcopy`, `pcopyin`, `pcopyout`, and `pcreate`: “`present_or_...`” the action only occurs if the data is not present
    - This distinction was made due to overhead concerns for the “present” check, but this concern turned out to be unwarranted

- Data management: how OpenACC gets your data to where it needs to be (GPU or CPU) and how you can steer this with a “data” construct.
  - By default, OpenACC automatically manages data movement between the CPU and GPU based on the data dependencies within the code
  - The “data” construct in OpenACC provides explicit control over data movement.
  - It allows the programmer to define data regions and specify how data should be treated in terms of movement between CPU and GPU.

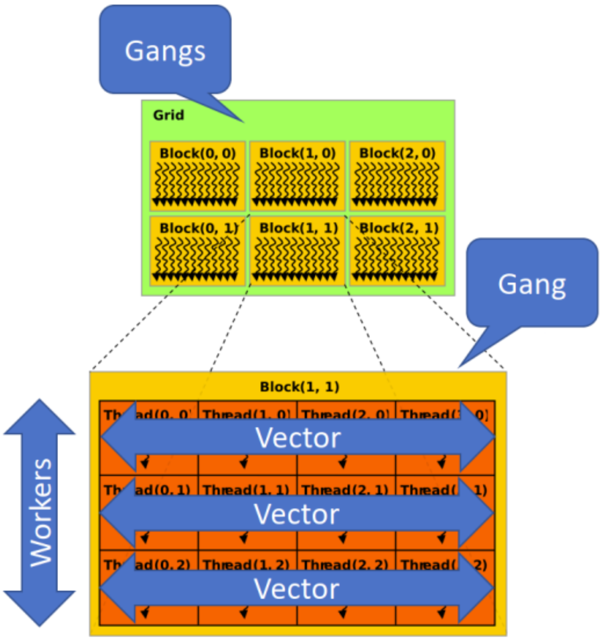
  ```
  #pragma acc enter data create(v[:n])
  #pragma acc exit data delete(v)
  ```

- Difference between “kernels” construct and “parallel” construct.
  - Kernel: OpenACC should do its best on that. Is very conservative.
  - Parallel: We have to know that it is correct. Out of order could break stuff.

- Difference between “grid”, “worker” and “vector”.
  - Grid: A higher-level organizational structure representing a collection of computing resources. It contains Gangs, which are equivalent to Blocks in Cuda.
  - Worker: An individual processing unit or entity responsible for performing a specific task. This is a collection of Vectors.
  - A gang is a pool of workers running on the same SM. This corresponds to a Cuda Block. Each SM can run at most 32 Gangs/Blocks.
  - Vector: A collection of Threads organized in contiguous memory, e.g. mapped to a Warp and executed at once.
  - General rule: Use vectors for the inner loop, gangs and workers for the outer. The maximum number of Blocks/Gangs per SM is 32. If we have a vector length of 32 and choose 32 blocks, we run 32*32=1024 Threads, which is half of the theoretical maximum of 2048 per SM. Then we would have a 50% occupancy
  - Occupancy (number of active threads) is a measure of how well threads handle latency. Occupancy is a indication of efficiency, but not a direct measure of it. We can use workers to help increase efficiency.

```
#pragma acc parallel vector_length(32)
#pragma acc loop gang worker
for(int i=0; i<n; ++i){
  #pragma acc loop vector
  for(int j=0; j<m; ++j){
    ...
  }
}
```

- Shared versus private variables.
  - Private by default: scalars and loop index variables, not like OpenMP
  - Shared by default: anything but scalars (i.e. arrays)
- Synchronization on the GPU, for example “reduce” clause, or “atomic” or “critical” pragmas. Performance of each.
  - An implicit synchronization happens when leaving a “kernels” or “parallel” region
  - There is no implicit synchronization inside a parallel region
    - The order of loop execution is not preserved - indexes are processed in any order
  - Loop execution order is preserved in the kernels construct
    - Each loop is a separate kernel invocation on the device
  - Critical: worst.
    - “implementing a barrier or critical section across workers or vector lanes using atomic operations and a busy-wait loop may never succeed, since the scheduler may suspend the worker or vector lane that owns the lock, and the worker or vector lane waiting on the lock can never complete.”
  - Reduce usually the fastest.
    - `#pragma acc parallel loop reduction(+:b)`
  - Atomic is ok, can have many waits if lots of writes on the same variable
    - `#pragma acc atomic [ read | write | update | capture ]`
    - If no clause is specified, the `update` clause is assumed.
- What asynchronous operations do and way you would want to use them.
  - Default: one parallel region at a time, but we can have multiple kernel launches at the same time.
  - The async(n) clause launches work asynchonously in queue n.
    - `#pragma acc parallel loop async(1)`
  - The wait(n) directive waits for all work in queue n to complete.
    - `#praqma acc wait(1)`


Gangs
Grid
Block(0, 0)
Block(1, 0)
Block(2, 0)
Block(0, 1)
Block(1, 1)
Block(2, 1)
Gang
Block(1, 1)
Thread(1, 0)
Thread(2, 0)
Vector
Thread(1, 1)
Thread(2, 1)
Vector
Thread(1, 2)
Thread(2, 2)
Vector
Workers

CUDA

- How to compile CUDA code (`nvcc`).
  - NVIDIA compiler (`nvcc`) separates host and device code
  - host code is passed to `gcc`
  - `nvcc hello_world.cu`
  - `__global__` keyword is used to indicate that a function runs on the device

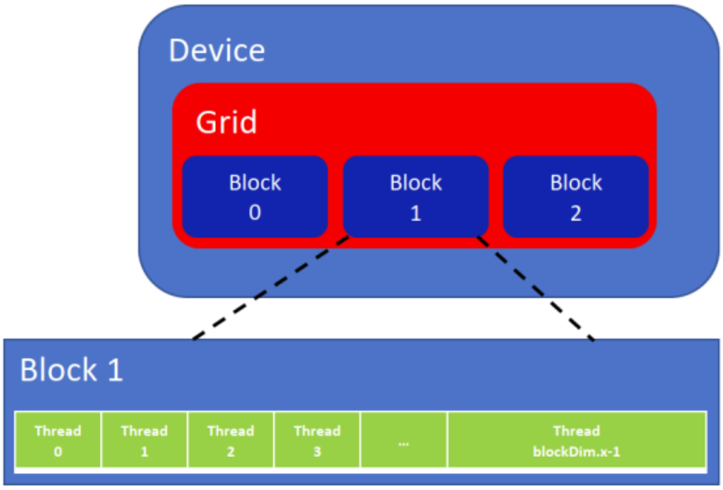
```
int size = 512*sizeof(int);
int a;                                                    //initialize host variables
int *a_d;                                                 //initialize device variables
cudaMalloc((void **)&a_d, size);                          //allocate storage on device
cudaMemcpy(a_d, &a, size, cudaMemcpyHostToDevice);        //copy to device
add<<<1,1>>>(a_d);                                        //launch kernel on the device
cudaMemcpy(&a, a_d, size, cudaMemcpyDeviceToHost);        //copy back to host
free(a);                                                  //free host variables
cudaFree(a_d);                                            //free device variables
```

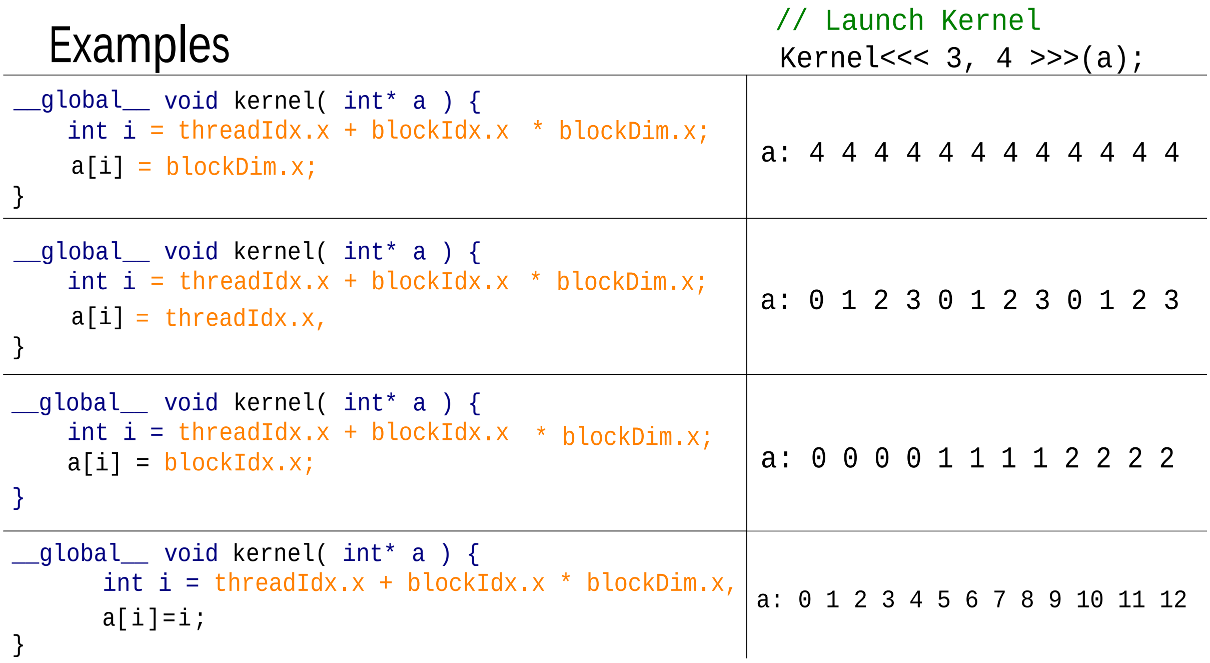
- Kernels:
- `name<<< Blocks per Grid, Threads per Block>>>(...);`
- Must be of return type `void`, they work by reference
- What a streaming multiprocessor (SM) is:
  - Each SM consists of multiple CUDA cores and various components necessary for parallel processing
  - CUDA cores are individual processing units within the SM that execute instructions independently
  - SMs have warp schedulers that manage the execution of warps. Warps are groups of threads that execute in lockstep. The scheduler ensures that warps are scheduled efficiently, maximizing parallelism
  - A GPU consists of multiple SMs working in parallel. The number of SMs varies depending on the GPU model and architecture. Usually they range from 32 to 128 SM's per Device.
- What is a “grid”, “block”, “warp” and “thread” and how they relate to the SM.
  - Grid:
    - A “grid” is a collection of blocks organized in a two-dimensional or three-dimensional structure. It represents the highest-level organizational unit in CUDA
    - The GPU's architecture allows multiple SMs to execute blocks from the grid concurrently. Grids are used to organize parallel tasks that require coordination between blocks
    - The kernel is run on a grid of blocks
  - Block:
    - A “block” is a group of threads that are scheduled to execute on an SM. Blocks are a higher-level organizational unit, grouping threads for parallel execution
    - An SM typically executes multiple blocks in parallel. Threads within a block can communicate and synchronize using shared memory, making it an essential unit for managing data sharing and synchronization
    - A Block runs on a single SM, and up to 32 Blocks at once per SM
  - Warp:
    - A “warp” is a group of threads (typically 32 threads) that execute the same instruction in lockstep on an SM
    - An SM schedules and executes warps. All threads within a warp execute the same instruction at the same time, facilitating efficient parallel processing
  - Thread:
    - A “thread” is the smallest unit of execution in GPU programming. It represents an individual task or operation that can be executed independently
    - Each thread within a warp performs the same instruction.
    - There is a limit on the number of threads per block. The limit is very small - usually 1024. (2*1024 : 2048 threads maximum per SM)

- How indexing works. How to turn a thread and block index into a global index.
  - threadIdx.x: Thread position inside the block
  - blockDim.x: Number of Threads per block
  - blockIdx.x: Block position index
  - gridDim.x: Number of blocks in the grid
    - `int index = threadIdx.x + blockIdx.x * blockDim.x`
    - Ceiling: `add<<<(N+Blocksize-1) / Blocksize, Blocksize>>>(..., N)` where N is an arbitrary vector length that is not necessarily a multiple of the Blocksize.
  - Example Pi Function:

```
// dx=1'000'000'000
// index <= 32'000
__global__ void cal_pi(double *sum, int dx, double step){
  int i;
  double x;
  int index = threadId.x + blockId.x * blockDim.x;
  for(i=index, i<dx, i+= blockDim.x * gridDim.x){
    x = (i+0.5)*step; //Calculation of the x-coordinate for each iteration
    sum[index] += 4.0/(1.0+x*x); //a numerical integration using the Monte Carlo method
  }
}
```

  - The For Loop Update `i += blockDim.x * gridDim.x` :
    - After each iteration of the loop, the loop variable `i` is increased by a certain stride
    - The stride is given by the product of the number of threads in a block (blockDim.x) and the number of blocks in the grid (gridDim.x). This gives us the total number of threads in the grid, we multiply the Number of Threads per block and the Number of blocks in the grid.
    - In effect, the entire grid of threads is jumping through the 1-D array of data, a grid-width at a time. This topic is sometimes called a "grid-striding loop"
    - Each thread processes a subset of the total iterations specified by `dx`, and the index is incremented by the total number of threads in the grid (blockDim.x * gridDim.x) in each iteration.
    - Rather than assume that the thread grid is large enough to cover the entire data array, this kernel loops over the data array one grid-size at a time
      - We can support any problem size even if it exceeds the largest grid size the CUDA device supports
      - By using a loop instead of a monolithic kernel, we can easily switch to serial processing by launching one block with one thread





Examples

```
// Launch Kernel
Kernel<<< 3, 4 >>>(a);
```

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x  * blockDim.x;
    a[i] = blockDim.x;
}
```

```
a: 4 4 4 4 4 4 4 4 4 4 4 4
```

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x  * blockDim.x;
    a[i] = threadIdx.x,
}
```

```
a: 0 1 2 3 0 1 2 3 0 1 2 3
```

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x  * blockDim.x;
    a[i] = blockIdx.x;
}
```

```
a: 0 0 0 0 1 1 1 1 2 2 2 2
```

```
__global__ void kernel( int* a ) {
      int i = threadIdx.x + blockIdx.x * blockDim.x,
      a[i]=i;
}
```

```
a: 0 1 2 3 4 5 6 7 8 9 10 11 12
```