

KAUST AMCS/CS 311
High Performance Computing I

Visiting Prof. Craig C. Douglas
University of Wyoming / KAUST

craig.c.douglas@gmail.com
<http://www.mgnet.org/~douglas>

<http://www.mgnet.org/~douglas/Classes/hpc-xtc>

Why Parallel Programming Is Necessary

All of your computers, pads, phones, etc. (will) have multiple cores. A core is just what we used to call a CPU (complete processors, just shrunk, with more than one on a silicon chip).

No serious speeding up programs otherwise in the foreseeable future.

- 1986-2002: microprocessors increased in speed $\sim 50\%$ /year, or a factor of ~ 57.66 in 10 years.
- 2003-present: microprocessors increased in speed $\sim 20\%$ /year, or a factor of ~ 6.2 in 10 years.
- Since 2005, there has been a huge design change in philosophy: parallel cores + no GHz speedup (in fact, expect slowdowns in the future).

- Compilers do not do well for parallelism, so programmers have to do the work. More cores means slower serial programs (more competition for memory and buses internal to the CPU).

Why do we care? Isn't 20%/year enough? No in these fields:

- Climate modeling and related areas: coupled models
- Protein folding (medical/pharmaceutical)
- Drug discovery (proteomics, genomics, chemical searching)
- Energy research (green energy)
- Data analysis (if you store it, you should use it)
- Many, many others

Why build parallel systems?

- Transistors can only get so small (smaller \Rightarrow faster), give off too much heat (faster \Rightarrow more power consumption \Rightarrow more heat), and wafers can only get so big
- New, messy physics and reliability issues in the electronics
- Current solution is multicore
- Go where no researcher has gone before

Why write parallel programs?

- Serial programs do not run on multiple cores and require a rewrite to parallelize (compilers are not good at doing the parallelization)
- Parallel algorithms are different from serial algorithms (special cases looked for by compilers with special code generated – table lookup process) and may not give the exact same result

- Completely new algorithms have been developed for many common problems.
- Adding cores is no help if programmers do not (know how to) use them

Simple example (C)

```
for ( i = sum = 0; i < n ; i++ )  
    sum += Compute_value( ... );
```

Watch for hidden dependencies in ... and Compute_value()!

Suppose we have p cores, labeled $0, 1, \dots, p-1$. We can divide the data into p parts of size $\sim n/p$. The parallel code has two parts: local and communications.

Consider the local part that each core can do independent of all other cores:

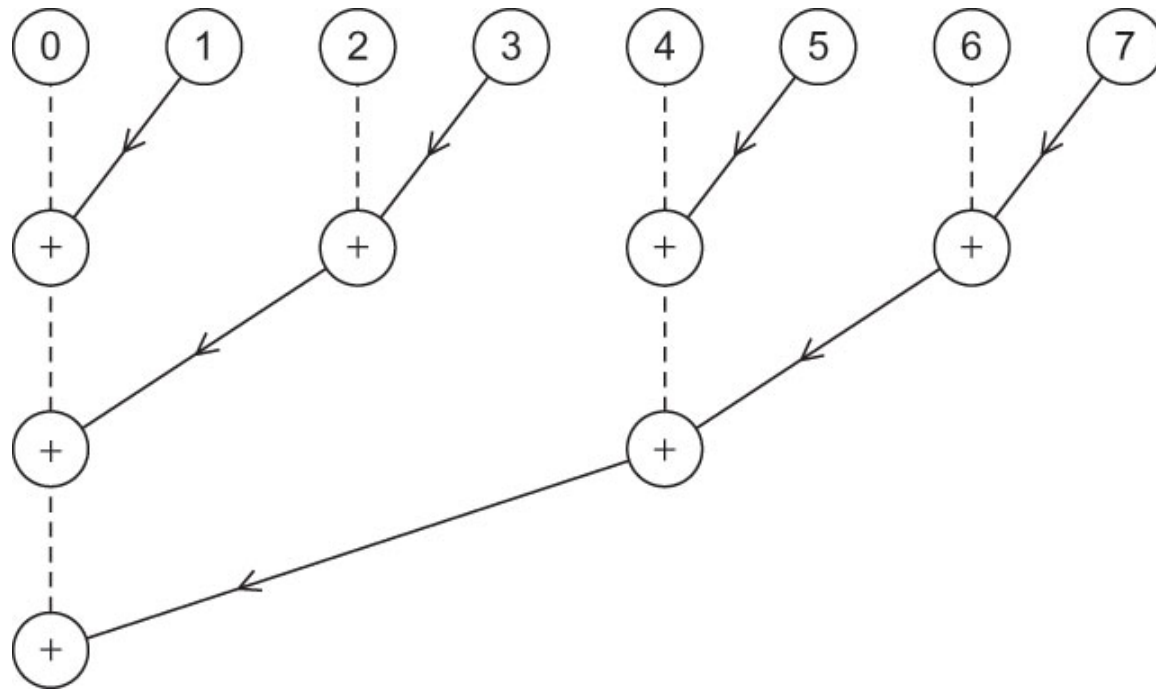
```
my_first_i = ...; my_last_i = ...;  
for ( my_sum = 0, I = my_first_i ; I < my_last_i ; i++ )  
    my_sum += Compute_value( ... );
```

Each core has a partial sum and the global sum has to be completed. We give two methods for the communication (global sum reduction).

Method 1 (sloooooooooooooow): requires $p-1$ communication/adds

```
if ( I'm core 0 ) {  
    sum = my_sum;  
    for ( core = 1 ; core < p ; core++ ) {  
        receive value from core;  
        sum += value;  
    }  
}  
else send my_sum to core 0;
```

Method 2 (tree, quicker): for a binary tree of $p = 8$ cores,



Now core 0 only has $\log_2 p$ receives and adds. In the example, 7 (method 1) versus 3 (method 2) is only about a factor of 2.

For $p = 1024$, $\log_2(1024)=10$, which is significantly less than 1023. Every time this p is multiplied by another factor of 1024, we add another 10 to the log result. When you are up to p in the trillions, e.g., $\log_2(1024^4)=40$, which is trillions less than method 1 would use.

The speedup comes from just using the communications network in a smarter manner, nothing else.

The log factor of 2 can be a larger integer and some systems adaptively change the log base based on how many CPUs (and racks of nodes) are being used and how far apart the racks are from each other.

What are the principles for writing parallel programs?

- Task-parallelism: partition what we want to do among the cores (tasks may or may not be the same)
- Data-parallelism: partition the data and work independently on local data (with communications and similar operations per core).
- Communications and load balancing:
 - Want to minimize communication (comm. => slow)
 - Want each core doing the same amount of work
 - Two points are usually not compatible and only an approximation works really on a computer
- Synchronization
 - Explicit: make all tasks wait for something
 - Implicit: communication or I/O that forces all tasks to wait for each other or do something collectively at the same time

Ambiguous Example: Grading homework with m homeworks, n questions, and $p \leq \min(n, m)$ graders

- Task-//ism: Each grader grades $\sim n/p$ separate problems on each homework
- Data-//ism: Each grader grades $\sim m/p$ separate homeworks, all problems
- Communication: accuracy versus communication with respect to people?
 - whole homeworks often in task-//ism
 - once per grader in data-//ism
- Load balancing: each grader has the same amount of work
- Synchronization: Asynchronous example, but there might want to be a discussion or similar (requires synchronization)

Here, there are lots of opportunities for random communication, completion times, and workflows. Similar to random walk algorithms (Monte Carlo algorithms) – nondeterminism of execution and all sorts of nasty behavior.

You need to decide what is a task. Is it grading a problem, part of all of a given problem, or an entire homework? Plan first, then devise the correct parallel scheme!

Simple Example (C) revisited

Data-//ism: on each core,

```
for ( i=my_sum=0 ; i < n ; i++ )  
    my_sum += Compute_value( ... );
```

Task-//ism: on all cores,

```
if ( I'm core 0 )  
    for( sum = my_sum, core = 1 ; core < p ; core++ ) {  
        receive value from core;  
        sum += value;  
    }  
else  
    send core 0 my_sum;
```

Two tasks: (1) adding and (2) receiving

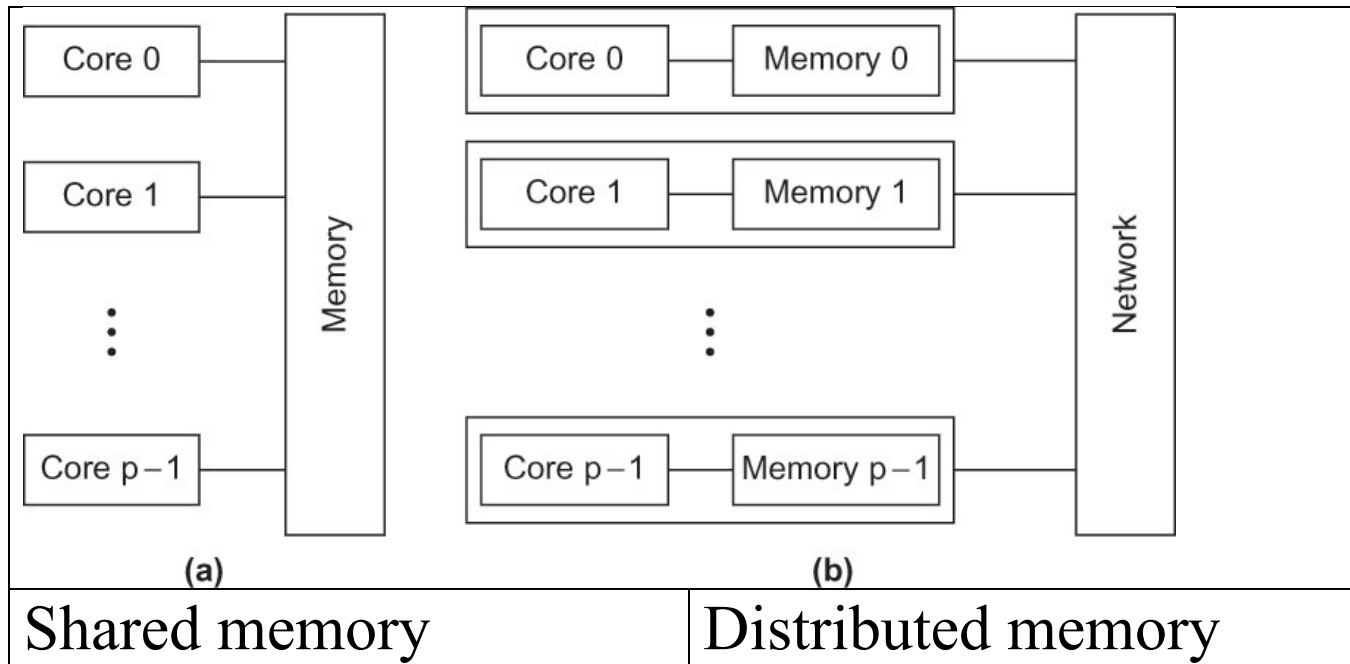
Coordination:

- Cores that have to coordinate their work
- Communication is done in sending partial sums to another core
- Load balance so that the work evenly distributed among the cores
- Synchronizing means leave no core too far behind

Categorization

- Typical communications methods
 - MPI (message passing interface) – distributed memory
 - OpenMP (or Pthreads) – shared memory
 - CUDA or OpenCL (or something like OpenMP pragmas, e.g., OpenAPP) – GP-GPUs, refrigerators, microwaves...
 - Combinations of first three bullets

- Types of systems
 - Shared memory
 - All cores can access all of memory
 - Coordinate cores using specific memory locations
 - Distributed memory
 - Each core has its own memory
 - Cores communicate explicitly by message passing
 - Nonuniform memory access (NUMA)
 - Communication of shared and distributed memory
 - Much more complicated than either model (often referred to as *multilevel* programming or the **lifetime employment act for parallel programmers**)



There are combinations of the above examples, too.

Terminology

- Concurrent computing
 - Any program that has multiple tasks in progress at any moment

- Parallel computing
 - Any program that has multiple tasks cooperating closely to solve a problem
- Distributed computing
 - All programs that may cooperate with other programs to solve a problem
- Parallel and distributed
 - Both are concurrent computing
 - No clear distinction, but parallel computers are usually physically close to each other.
 - Cloud, GRID, ... computing can be either parallel or distributed computing and are usually concurrent computing.

Programming advice

- Writing parallel programs without careful planning in advance is a disaster, waste of time and money, and a great way to take an unplanned holiday after termination
- Usually a serial program hiding in the parallel code. Great care to work around and with the serial code. It becomes the big bottleneck.
- Multiple cores are harder to coordinate than 1 core. Programs are much more complex. Algorithms are trickier and prove they will always work correctly. Answers are not always the same bitwise, run to run, or in comparison to serial.
- Good programming habits are far more important in parallel programming than in serial programming.
- Always document what you code very carefully:
 - Specify algorithms (include details)

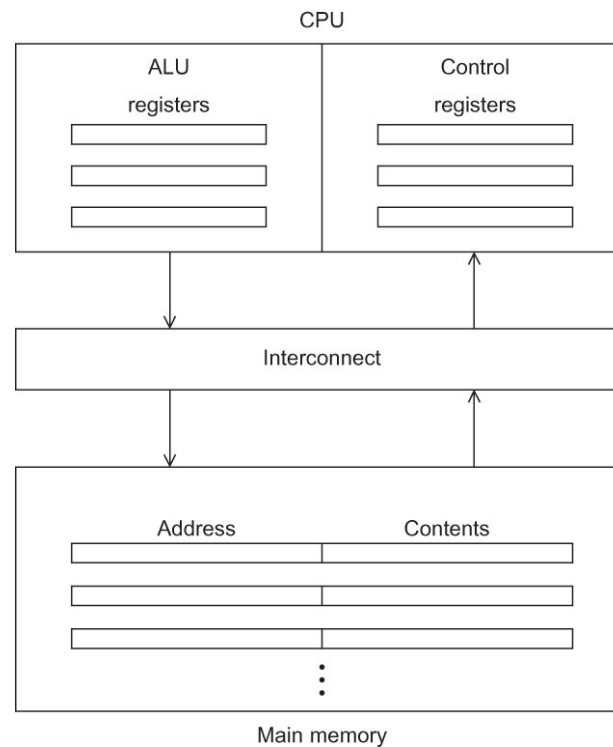
- Provide easy to find citations (conference proceedings do not count – complete waste of time to try to find) 😊
- If someone else cannot figure out your code, then you probably will not be able to either in six months. ☹

Hardware

Von Neumann model (1940's) – 4 parts

- Main memory
 - Many locations each with an address that can store both data and instructions
- CPU (1 core today)
 - Control unit (CU) decides which instructions to execute and their order (branching decisions, etc.)
 - Arithmetic and logic unit (ALU) executes instructions
 - Registers are (very) fast memory locations in ALU
 - Program counter register in CU contains the address of the next instruction for execution
- Interconnect between main memory and CPU

- Traditionally a collection of parallel wires called a bus plus logic to run it
- The **von Neumann bottleneck** is this interconnect
- One instruction executed at a time on only a few pieces of data



Terminology

- Data and instructions are read or fetched from memory
- Data is stored or written to memory
- In 1940's, data movement/instruction execution was ≤ 1 .
Today it is ≥ 100 . A big change occurred in the 1990's
- Today for communication between distributed CPUs, it is $\geq 1,000$ and sometimes $\geq 20,000$

Von Neumann model extensively modified over time

- Almost nothing left untouched in any component
 - Main memory first (1960's)
 - CPUs next (also 1960's)
 - Parallel computers studied extensively, first in theory (early 1960's onward), then were built (late 1960's)

- CPU-memory interconnects (1980's and 1990's were the golden era)
- Von Neumann would not recognize his model today

What is a process?

- Created by operating system
- Executable program (typically in machine language)
- Contains a block of memory
 - Executable code
 - Call stack (for active function usage)
 - Heap (dynamic memory arena)
- Security information (e.g., what hardware/software can be accessed)
- State information
 - Ready to run or waiting for something information

- Register contents when blocked
- Process memory information

Multitasking

- Appears that a single processor (core) can run multiple programs at once
- Time slicing: each process runs and then is blocked (either because of a resource request or too much time used). Waits until given another time slice

Threading (regular and light weight threads)

- Multitasking within a process
- Each thread independent of other threads in a process
- Shares process resources (e.g., memory)
- Hopefully allows a process to use all of each time slice

OpenMP and Threads

Threads are a concept, but have been implemented first as software, then in hardware. Today, a combination of the two forms provides very fast switching between threads in a process.

Threads are not true parallel computing objects. Only one thread runs at a time on a core. A core that allows hardware multi-threading has additional hardware (e.g., registers and state information) to switch between threads in very little time with very little overhead (the hardware duplicated does not have to be exchanged with memory backups to do the thread switch).

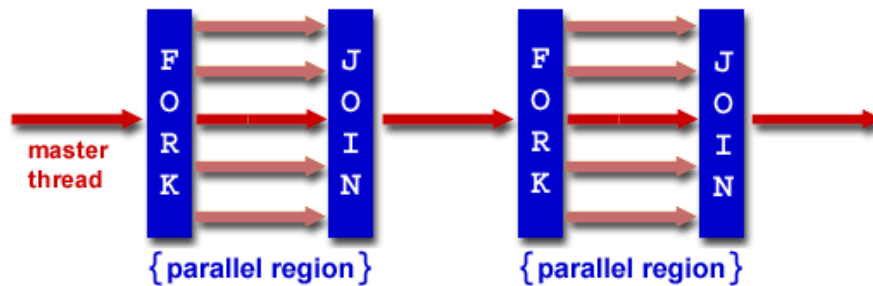
OpenMP:

- *Open Multi-Processing* (see <http://www.openmp.org>, <http://en.wikipedia.org/wiki/OpenMP>, and the links therein)
- Allows multi-threaded, *shared* memory *explicit* parallel programming
- Meant for coarse grained (possibly nested) parallelism based on an original serial code
- Is a three fold *portable* system consisting of
 - Compiler directives (C/C++/Fortran)
 - A runtime library API
 - Environment variables
- Scalable up to some number of cores per shared memory
- Is standardized by compiler groups (ANSI/ISO someday)
- An open specification with a managing group

OpenMP is *not*:

- Meant for distributed memory, parallel systems (combine it with MPI in this case)
- Implemented identically by all vendors (some exotic parts not always implemented)
- The most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization (and can cause conflicts without care)
- Designed to guarantee that input or output to the same file is synchronous when executed in parallel. *The programmer is responsible for synchronizing input and output.*

OpenMP uses the Fork-Join model:



- All programs start with 1 *master* thread (thread 0).
- When the first parallel region is encountered, the system fork() routine is called to create p-1 extra threads (*team* of p threads).
- The parallel region runs in a random thread order and time slicing.
- The extra threads either exit or are put to sleep in the Join step.
- The master thread continues running alone again.

Inconsistencies over the years: not all implementations have had these features, but all should by now (famous last words ☹).

- Dynamic threading means you can change the number of threads at will inside the running code.
- Parallel I/O
 - Nothing specified in OpenMP (programmer problem).
 - If each thread reads/writes to a separate file, all okay.
 - Accesses of one file by multiple threads has to be coordinated by the programmer explicitly.
- Memory consistency
 - Each thread can have a private copy of a variable.
 - If there is a need for consistency on a global basis, it is up to the programmer to flush the local value back to the global value and make sure that the global value is consistent.

OpenMP Directives

Examples:

C/C++:

```
#pragma omp parallel  
# pragma omp parallel for
```

Fortran (some are in pairs):

```
!$omp parallel  
...  
!$omp end parallel
```

- No comments on a directive line
- One directive name per line
- Some directives are really compounds

```
#pragma omp parallel \  
    if (scalar expression) \  
    private( list ) shared( list ) default( shared | none ) \  
    firstprivate( list ) reduction( operator: list ) copyin( list ) \  
    num_threads( integer expression )
```

- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team.
- The code in the parallel region is duplicated. All threads execute that code.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

Example: Trivial hello program

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main( int argc, char** argv ) {

    // Master thread
    int nthreads = strtol( argv[1], NULL, 10 );

    // Parallel threads
    #pragma omp parallel num_threads( nthreads )
    printf( "Hello from thread %d\n", omp_get_thread_num() );

    // Join back to master thread
    return 0;
}
```


Restrictions:

- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch into or out of a parallel region
- Only a single IF clause is permitted
- Only a single NUM_THREADS clause is permitted

How many threads in a PARALLEL section?

1. Evaluation of the IF clause
2. Setting of the NUM_THREADS clause
3. Use of the `omp_set_num_threads()` library function
4. Setting of the `OMP_NUM_THREADS` environment variable
5. Implementation default - usually the number of CPUs on a node, though it could be dynamic.

Dynamic threading?

- `omp_get_dynamic()` will tell you if it is available.
- If it is, use `omp_set_dynamic()` or set `OMP_DYNAMIC` environment variable to `TRUE`.

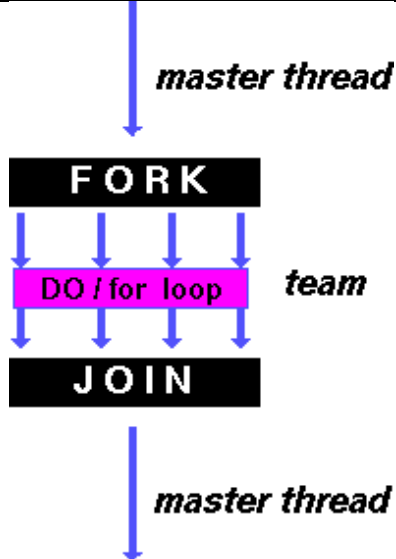
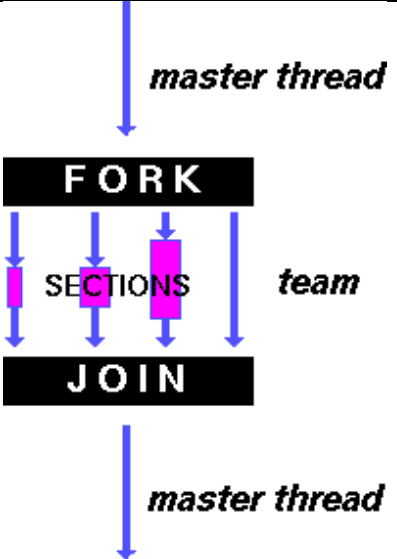
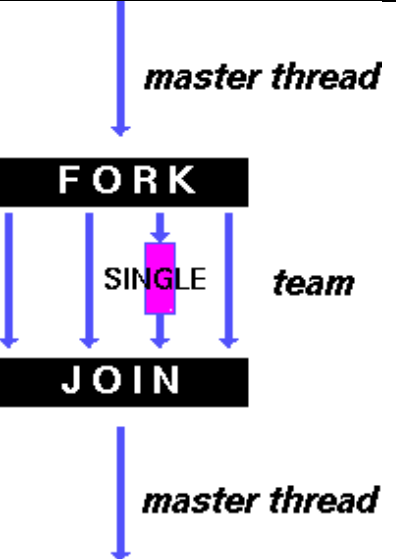
Nested threading?

- Use the `omp_get_nested()` library function to determine if nested parallel regions are enabled.
- If it is, use `omp_set_nested()` or set `OMP_NESTED` environment variable to `TRUE`.
- If it is not, a parallel region nested within another parallel region results in the creation of a new team, consisting of one thread, by default.

Work-sharing constructs

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it into
 - Data parallelism
 - Functional parallelism
 - Serial
- Work-sharing constructs do not launch new threads, but use the current ones.
- No implied barrier upon entry to a work-sharing construct, but an implied barrier at the end of a work-sharing construct.

Examples:

 <p>The diagram shows a master thread (blue arrow) entering a FORK block. From the FORK block, four parallel blue arrows lead to a pink box labeled DO / for loop. To the right of this box is the word <i>team</i>. Four parallel blue arrows then lead from the pink box to a JOIN block. Finally, a single blue arrow exits the JOIN block, labeled <i>master thread</i>.</p>	 <p>The diagram shows a master thread (blue arrow) entering a FORK block. From the FORK block, four parallel blue arrows lead to three separate pink boxes, each labeled SECTIONS. To the right of these boxes is the word <i>team</i>. Four parallel blue arrows then lead from the pink boxes to a JOIN block. Finally, a single blue arrow exits the JOIN block, labeled <i>master thread</i>.</p>	 <p>The diagram shows a master thread (blue arrow) entering a FORK block. From the FORK block, four parallel blue arrows lead to a single pink box labeled SINGLE. To the right of this box is the word <i>team</i>. A single blue arrow then leads from the pink box to a JOIN block. Finally, a single blue arrow exits the JOIN block, labeled <i>master thread</i>.</p>
Data parallelism: shares iterations of a loop across the team.	Functional parallel- ism: breaks work into separate, discrete sections with each executed by a thread.	Serializes a section of code.

Work-sharing considerations and restrictions:

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- There is neither a guarantee in the order of thread execution nor number of time slices to completion.
- All members of a team or none must encounter work-sharing constructs. No partial thread encounters occur.
- All members of a team must encounter successive work-sharing constructs in the same order.
- (Somewhat tricky English here.)

Data handling directives

There are 8 directives for data scoping that are used through OpenMP's work-sharing directives.

- Unless specified, OpenMP's variables all lie in shared memory.
- Shared data usually includes static and file scope variables.
- Private variables should usually include loop indices, automatic variables inside a parallel block, and stack variables from a called subroutine.
- This mechanism defines how variables are treated at the beginning, during, and at the end of a PARALLEL, FOR/DO, or SECTIONS block of OpenMP code (which ones are visible at a given time and which ones are not).
- *Only* apply to current OpenMP block of code.

`private(list)`

- Each thread gets its own copy, which is uninitialized.
- Value lost at end of block.

`shared(list)`

- Each thread uses the same memory location, which retains its value from before the block.
- Value persists after the end of block.
- Programmer must ensure that only one value occurs at a time through CRITICAL sections if necessary.

`firstprivate(list)`

- Listed variables have their own copy and are initialized with the value from before the block.

`lastprivate(list)`

- The value obtained from the last (sequential) iteration or section of the enclosing construct is copied back into the original variable object.

`default(shared | none)`

- All remaining variables have the given property.
- None means all variables have to be given a property.

`copyin(list)`

- The master thread variable is used as the copy source. All team threads are initialized with its value upon entry into the block.

copyprivate(list)

- Used with the SINGLE directive for serialization of a team.
- This clause is used to broadcast values from a single thread directly to all instances of the private variables in the other threads.

reduction(operation: list)

- This is used to do a reduction operation, e.g., an inner product with a scalar result from two shared global vectors so that operation would be + in this case.
- A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable and the final result is written to the global shared variable.

The actual operations allowed in a reduction are the following:

$x = x \text{ op } \text{expr}$

$x = \text{expr op } x$ (except subtraction)

$x \text{ binop} = \text{expr}$

$x++$

$++x$

$x--$

$--x$

More specifically,

- x is a scalar variable in the list
- expr is a scalar expression that does not reference x
- op is not overloaded, and is one of $+$, $*$, $-$, $/$, $\&$, $^$, $|$, $\&\&$, $||$
- binop is not overloaded, and is one of $+$, $*$, $-$, $/$, $\&$, $^$, $|$

Summary of clauses and directives

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Work-share construct *for* (C/C++) or *do* (Fortran)

```
#pragma omp for [clause ...] \  
    schedule (type [,chunk]) ordered \  
    private (list) firstprivate (list) lastprivate (list) shared (list) \  
    reduction (operator: list) collapse (n) \  
    nowait  
    for loop
```

schedule

- *chunk*: integer expression giving granularity of piece of loop to assign at any given time to a thread.
- *auto*: The compiler and/or runtime system decides everything.
- *static*: fraction of loop assigned to a thread is size *chunk* or is evenly split.

- *dynamic*: threads run a chunk sized piece of the loop and then receive another chunk-sized piece until the loop is completed (default chunk size = 1).
- *guided*: For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size $k > 1$, the size of each chunk is determined similarly, but the chunks do not contain fewer than k iterations except for the last chunk assigned that may have fewer than k iterations (default chunk size = 1).
- *runtime*: The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. A chunk size for this clause cannot be specified.

nowait

- Threads do not synchronize at the end of the loop.

ordered

- Iterations of the loop must be done in the order of a serial loop.

collapse

- Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

Notes:

- No branching out of a FOR block.
- Implied barrier at end of a FOR block.

Example: Inner product of 2 vectors

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

main () {
    int    i, n = 100, chunk = 10;
    double a[100], b[100], result = 0.0;

    for( i=0; i<n; i++ ) {
        a[i] = i;
        b[i] = i * 1.5;
    }

    #pragma omp parallel for default(shared) private(i) \
                        schedule(static,chunk) \
                        reduction(+:result)

    for( i=0; i < n; i++ )
        result = result + (a[i] * b[i]);

    printf("Final result= %le\n",result);
}
```

Work-share constructs *sections* and *section* (C/C++)

- The SECTIONS directive is a non-iterative work-sharing construct that specifies that the enclosed section(s) of code must be divided among the threads in the team.
- Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team.
- It is possible for a thread to execute more than one SECTION if it is quick enough and the implementation permits such.
- There is no guarantee what order each SECTION will be run.
- If there are dependencies between more than one SECTION, it is the programmer's responsibility to make them work correctly. Inter SECTION dependencies should be avoided if possible.


```
#pragma omp sections [clause ...] \  
    private (list) firstprivate (list) lastprivate (list) \  
    reduction (operator: list) nowait  
{  
#pragma omp section  
    structured block  
  
#pragma omp section  
    structured block  
}
```

Notes:

- No branching out of a SECTION block.
- Implied barrier at end of SECTIONS block.

Example: Different operations on 2 vectors

```
#include <omp.h>

main () {
    int i, N = 1000;
    float a[N], b[N], c[N], d[N];

    /* Some initializations */
    for (i=0; i < N; i++) {
        a[i] = i;
        b[i] = i * 1.5;
    }

    #pragma omp parallel shared(a,b,c,d,N) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i<N; i++)
                c[i] = a[i] + b[i];

            #pragma omp section
            for (i=0; i<N; i++)
                d[i] = a[i] * b[i];
        } /* end of section collection */
    } /* end of parallel sections */
}
```

Work-share construct *single* (C/C++)

- The SINGLE directive specifies that the enclosed code must be run by exactly one thread in the team.
- It is useful when dealing with sections of non-thread safe code (e.g., I/O).
- Unless a `nowait` clause is specified, threads in the team that do not execute the SINGLE directive wait at the end of the enclosed code block.

```
#pragma omp single [clause ...]  
    private (list) firstprivate (list) nowait  
    structured block
```

Note:

- No branching out of a SINGLE block.

Work-share/multitasking construct *task* (C/C++) – OpenMP 3.0+

- Defines an explicit task that is run by the encountering thread or deferred and run by any other thread in the team.
- Depends on task scheduling in OpenMP 3.0 and beyond.

#pragma omp task [clause ...]

if (scalar expression)

untied default (shared | none)

private (list) firstprivate (list) shared (list)

structured block

Synchronization construct *master* (C/C++)

- Only runs on the master thread.

`#pragma omp master`
structured block

Note:

- No branching out of a MASTER block.

Synchronization construct *critical* (C/C++)

- Only one thread at a time runs this block of code.
- Multiple different CRITICAL regions can exist: names act as global identifiers. Different CRITICAL regions with the same name are treated as the same region (unnamed regions are considered the same region).

#pragma omp critical [name]
structured block

Example: Incrementing a global variable

```
#include <omp.h>

main() {
    int key = 0;

#pragma omp parallel shared(key)
    {
        ... /* thread code goes here */
#pragma omp critical
        key++;
    }
    ... /* more thread code goes here */
}
```

Synchronization construct *atomic* (C/C++)

- A specific memory location must be updated atomically rather than letting multiple threads attempt to write to it.
- Only one statement follows this construct.

```
#pragma omp atomic  
    statement expression
```

Example: Modification of CRITICAL section example.

```
#pragma omp atomic  
    key++;
```


Synchronization constructs *barrier* and *taskwait* (C/C++)

- BARRIER synchronizes all threads in the team.
- TASKWAIT waits for the completion of child tasks generated since the beginning of the current task.
- Care in placement is required since no code comes after the #pragma statement.

#pragma omp barrier

#pragma omp taskwait

OpenMP runtime functions

```
#include <omp.h>
```

This include file is necessary for all files using OpenMP.

Threads/Processors:

```
int num_threads, max_threads, thread_num, thread_limit, proc_max;  
  
omp_set_num_threads(num_threads); // Set default number  
num_threads = omp_get_num_threads();  
max_threads = omp_get_max_threads();  
thread_num = omp_get_thread_num(); // My thread number  
thread_limit = omp_get_thread_limit(); // OpenMP 3.0+  
  
proc_max = omp_get_num_procs();
```

Examples:

```
#include <omp.h> // Correct
```

```
...  
int tid;  
#pragma omp parallel private( tid )  
tid = omp_get_thread_num();
```

```
# include <omp.h> // Does not work as expected
```

```
...  
int tid;  
#pragma omp parallel  
tid = omp_get_thread_num();
```

```
# include <omp.h> // Does not work
```

```
...  
int tid;  
tid = omp_get_thread_num();
```

Parallelism:

```
if ( omp_in_parallel() ) { running in parallel }  
if ( omp_get_dynamic() ) { dynamic threading available }  
if ( omp_get_nested() ) { nested threading available }
```

```
omp_set_dynamic(int dynamic_threads);
```

```
omp_set_nested(int nested_threads);
```

- Must be in a serial part of code.
- Might enable/disable dynamic or nested threading (system dependent).
- Can fail and not tell you!

Timing:

```
double t1, tt;
```

```
t1 = omp_get_wtime();
```

```
tt = omp_get_wtick();
```

Provides portable CPU time per thread. Wtime provides a time in seconds. Wtick provides timer ticks. Use `omp_get_wtime()` or a wall clock timer (which is what you really want, not CPU time).

Locks:

Declarations for lock routines below.

```
void omp_init_lock(omp_lock_t *lock) void  
omp_init_nest_lock(omp_nest_lock_t *lock)
```

- Initial state for a lock variable is *unlocked*.

```
void omp_destroy_lock(omp_lock_t *lock) void  
omp_destroy_nest_lock(omp_nest_lock_t *lock)
```

```
void omp_set_lock(omp_lock_t *lock) void  
omp_set_nest_lock(omp_nest_lock_t *lock)
```

```
void omp_unset_lock(omp_lock_t *lock) void  
omp_unset_nest_lock(omp_nest_lock_t *lock)
```

```
int omp_test_lock(omp_lock_t *lock) int  
omp_test_nest_lock(omp_nest_lock_t *lock)
```

Environment variables (runtime routines can override):

* OpenMP 3.0+

OMP_SCHEDULE

OMP_NUM_THREADS

OMP_DYNAMIC

OMP_NESTED

OMP_WAIT_POLICY

OMP_STACKSIZE*

OMP_MAX_ACTIVE_LEVELS*

OMP_THREAD_LIMIT*

Compiler flags to turn on OpenMP:

Compiler	Flag
GNU	<code>-fopenmp</code>
Intel	<code>-openmp</code>
IBM	<code>-qsmp=omp</code>
PGI	<code>-mp</code>

Compiling and running with gcc:

```
gcc -fopenmp file1.c ... fileN.c -o file.exe
```

```
./file.exe [args]
```


Memory and Memory Programming

Primary memory (from smallest/fastest to largest/slowest):

- Registers (in ALU on core)
- L1 cache(s) (usually on CPU and next to core(s))
- L2 cache (frequently on CPU)
- L3 cache (usually not on CPU)
- L4 cache (almost never anymore)
- Main memory (separate chip(s))

Secondary memory:

- Disk (near CPU or on network)
- Tape (usually on network)

On a cluster, there is usually some local disk storage. This is as fast as on a typical PC. Your home directory is not on a local disk.

Principles of locality:

- Spatial: accesses occur nearby the current access.
- Temporal: accesses occur again real soon now.

Virtual memory

- For programs with large enough data sets, not everything fits into the physical main memory.
- Virtual memory acts like a cache system for secondary storage.
- Uses both principles of locality.
- Only the active parts of the running program and data are in the physical main memory (with a complicated mapping).
- Swap space: idle parts of program and data are on disk.
- Pages:
 - Blocks of program instructions or data.
 - Most systems have a fixed page size of 4-16 KB.

- Virtual page numbers:
 - Pages have a virtual number assigned to them.
 - At run time, a table is created that maps virtual page numbers to physical pages and is dynamic.
 - The page table is used by hardware to run the memory system and is chaotic.
 - It is very difficult, if not impossible, to reproduce the table when running the application repeatedly.
- Translation Look-aside Table (TLB)
 - A special address translation cache in the processor to significantly speedup page table references.
 - Caches 16-512 page table entries in very fast memory.
- Page fault
 - When a page on disk is addressed, a page has to be moved into main memory and one removed or written to disk.

Example: Why should we care about memory levels?

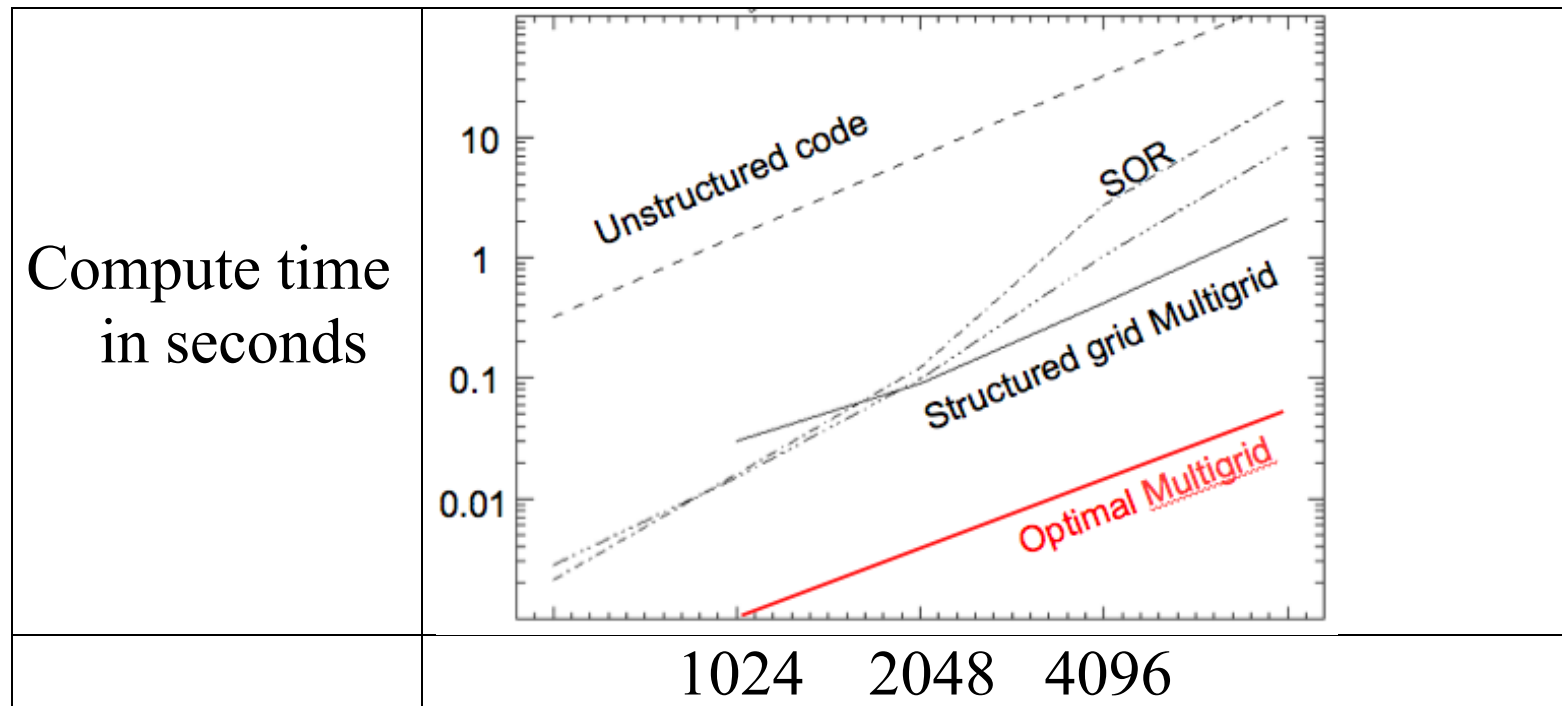
- Let's solve Poisson's equation on a square.
 - Multigrid algorithm solves it in ~ 28 Flops per unknown.
- Let's solve more general elliptic PDE problems.
 - Multigrid algorithm solves it in $O(100)$ Flops per unknown.
- High end laptop today can compute ~ 10 GFlops
 - Poisson should solve 3 B unknowns per second.
 - General elliptic should solve 1 B unknowns per second.
 - Amount of memory is about $4C/3$ the number of unknowns, where C is $O(1)$ and problem dependent.
- How fast and reliable are real solvers?
 - No more than 10-100 K unknowns before code breaks.
 - In time of minutes to hours, not fractions of a second.
 - Use horrendous amounts of memory that was not expected.
 - *Even state of the art codes are usually very inefficient!*

- Revised assumptions:
 - Multigrid takes 500 Ops/unknown to solve your favorite PDE.
 - You can get 5% of 10 Gflops performance.
 - On your 10 GFlop laptop you will solve only one million unknowns in 1.0 second, i.e., ~1 microsecond per unknown instead of 3-10 nanoseconds per unknown. ☹️

Ideal versus Real Performance

- For decades, counting Flops was the measure in modeling performance of numerical algorithms.
- Measuring memory performance is now the correct measure, but it is much harder to model due to chaotic behavior of caches and virtual memory.
- Never, *ever* use virtual memory if you want high performance.

What partly got me started with cache memory aware algorithms and implementations in the early to mid 1990's:



Example of a possible linear speedup:

```
double w[n], x[n], y[n], z[n];  
for( int i = 0; i < n; i++ )  
    w[i] = x[i] + y[i] + z[i];
```

versus

```
double arrays[n][4];  
for( int i = 0; i < n; i++ )  
    arrays[i][0] = arrays[i][1] + arrays[i][2] + arrays[i][3];
```

For a cache line of 32 bytes (4 doubles), when the first version has a cache miss, it usually has 4 cache misses and there is no cure. For the second version, compilers will try to prefetch cache lines and there is at most one cache miss when one occurs, not four.

How does a cache work?

- Moves a cache line of 32-128 bytes, not a word of 4-8 bytes.
- When a cache line is changed, then main memory is inconsistent and will need to be updated eventually.
 - Write through caches do the writing in the background when a memory bus is available.
 - Write back caches do the writing only when a cache line is expunged from cache.
- Core waits or freezes until
 - Bytes in cache, or
 - Cache line is in cache.
 - May require a cache line to be written back to main memory and writing times takes 1-2 times reading time from main memory.

- Where does a cache line go in the cache?
 - Direct mapped means a physical memory address maps to exactly one place in the cache (a very bad idea because of possible cache line thrashing).
 - Fully associative means a new cache line can go anywhere in the cache (a bad idea because it costs too much to build).
 - N-way associative means each cache line can go in exactly N different places in the cache (standard methodology as a compromise)
 - An interesting question is which of the N places would be picked and can require knowledge about which line will be expunged from cache to fit the incoming line.

Current CPU characteristics

CPU:	AMD 6140	Intel i7-2600K	Intel Atom D525	Intel E7-8870
Cores	8	4	2	10
Threads/core	4	2	2	2
L1 cache	2 x 64 KB / core	2 x 32 KB / core	32 KB data, 24 KB instruction	2 x 32 KB / core
L2 cache	8 x 512 KB	4 x 256 KB	1 MB	10 x 256 KB
L3 cache	2 x 6 MB	8 MB	n/a	30 MB
Memory channels	3	2	1	4

AMD 6180 similar to the 6140, but with 12 cores (12 L2 caches).

8-way associative L2/L3 caches.

Classic **Memory Wall** Problem

Latency: time for memory to respond to a read (or write) request is too long since

- CPU ~ 0.5 ns (light travels 15cm in vacuum)
- Memory ~ 50 ns

Bandwidth: number of bytes that can be read or written per second

- CPUs with C GFLOPS peak performance need $24C$ GB/sec bandwidth.

How do you break through the memory wall???

What does a cache look like electronically (e.g., Motorola 68020)?
(figures from J. Handy book, 1998)

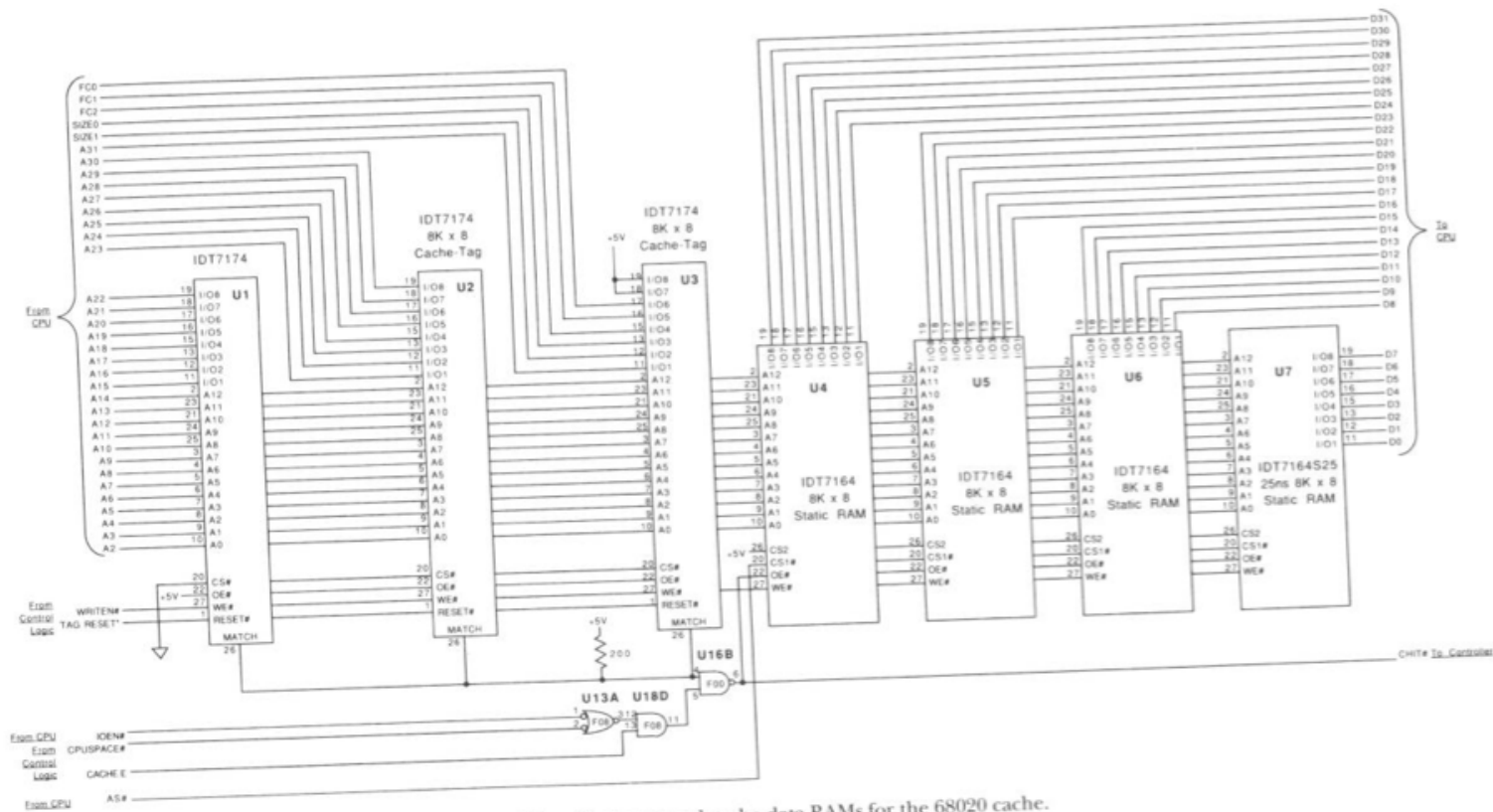


Figure 1.15a. Cache-tag and cache data RAMs for the 68020 cache.

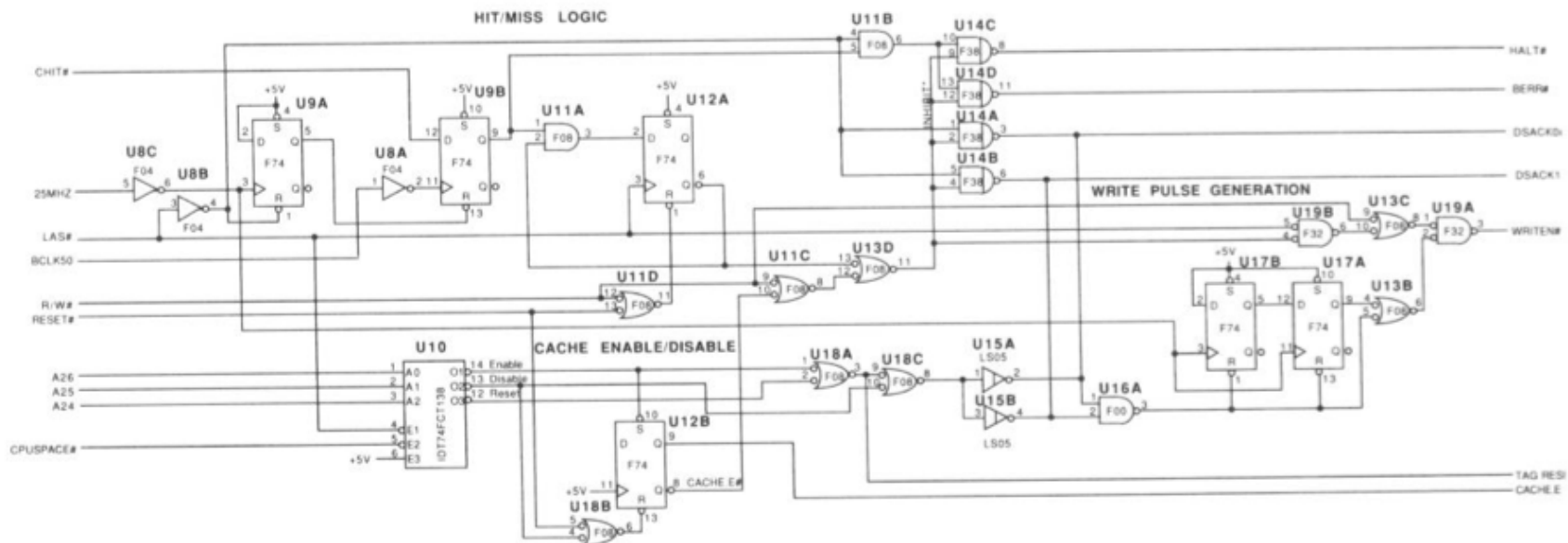
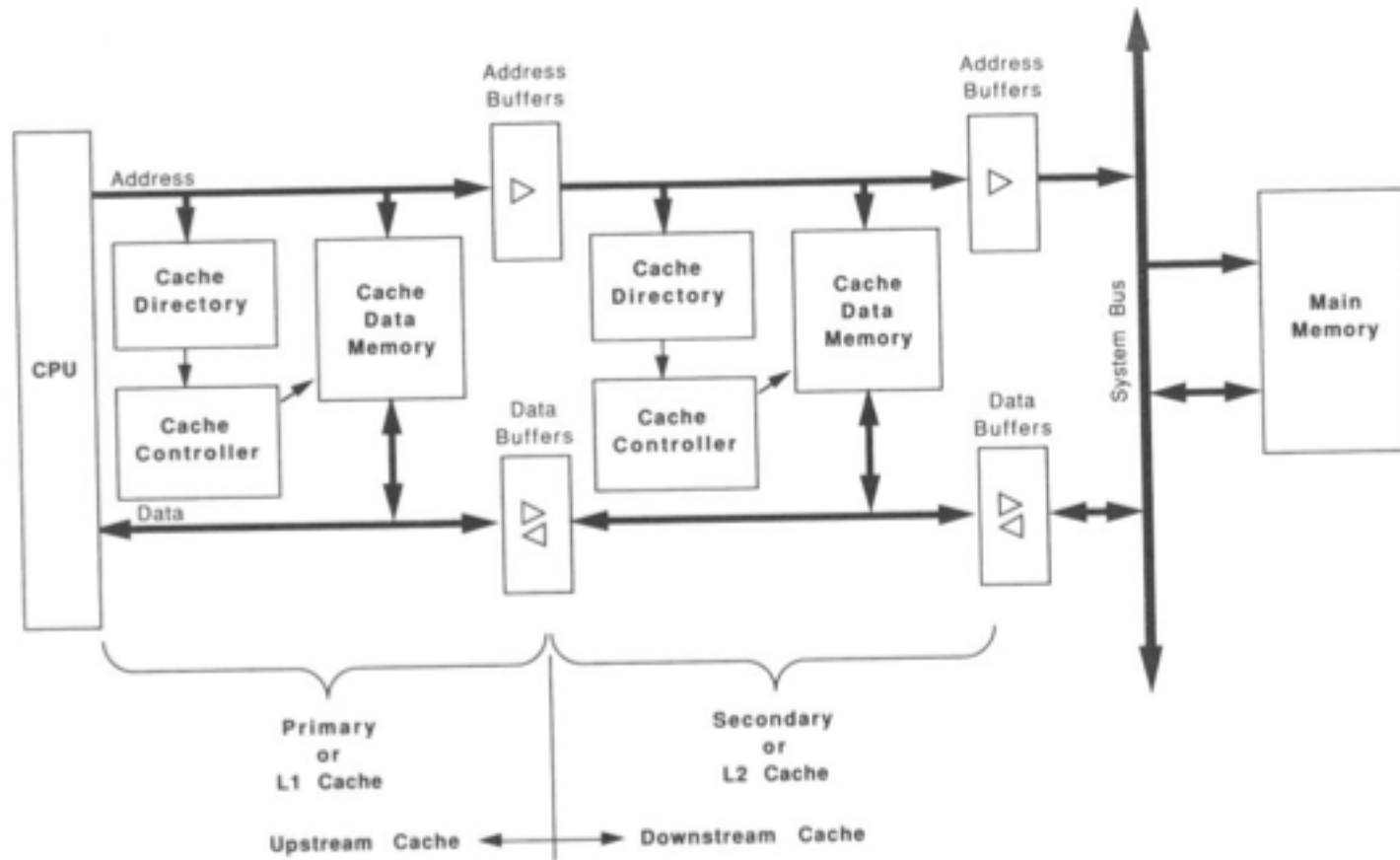


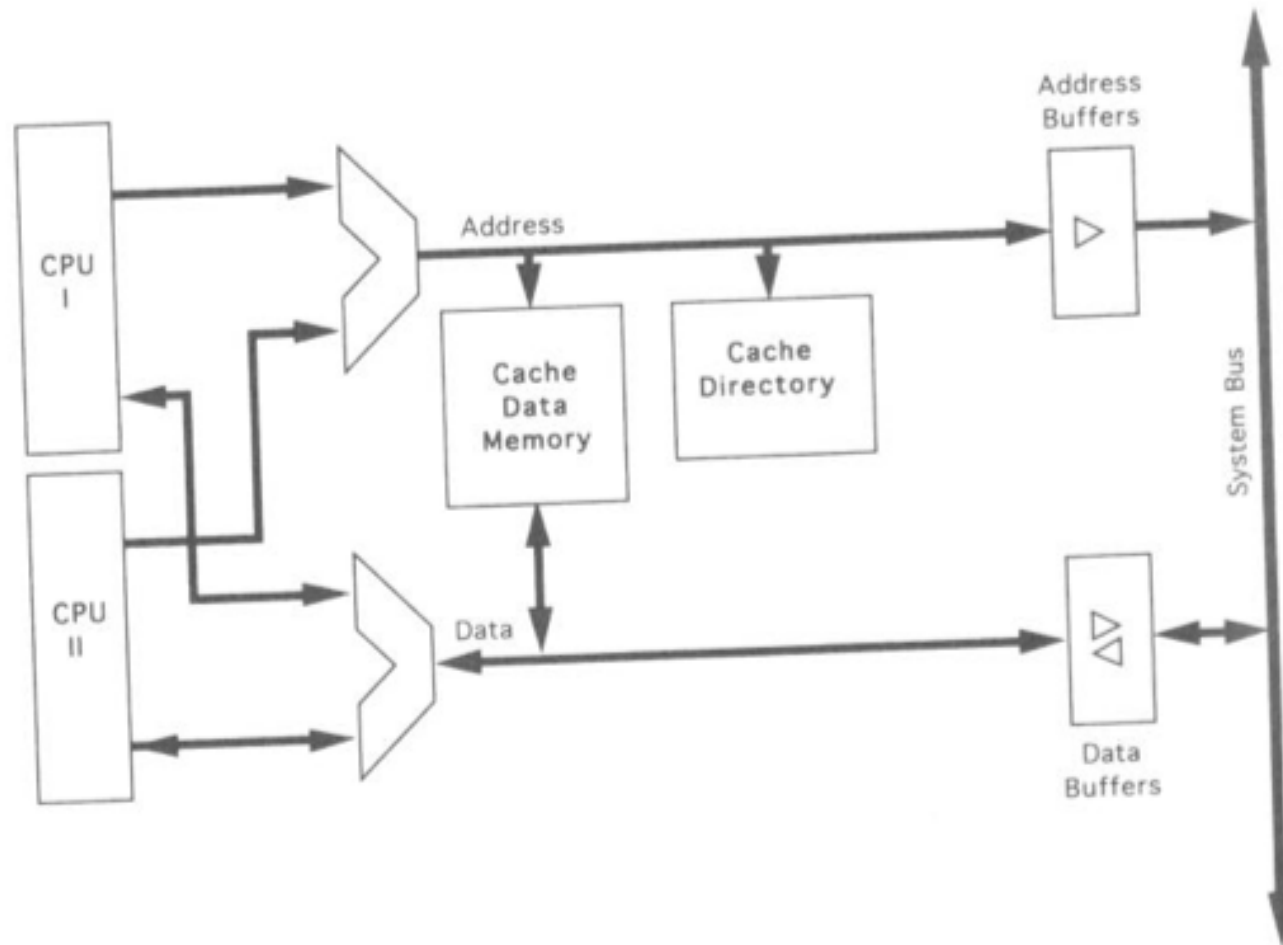
Figure 1.15b. Logic of cache controller for 68020 cache.

The 68020 cache system was quite simple (circa 1983). Today cache schematics take up a small book in complexity.

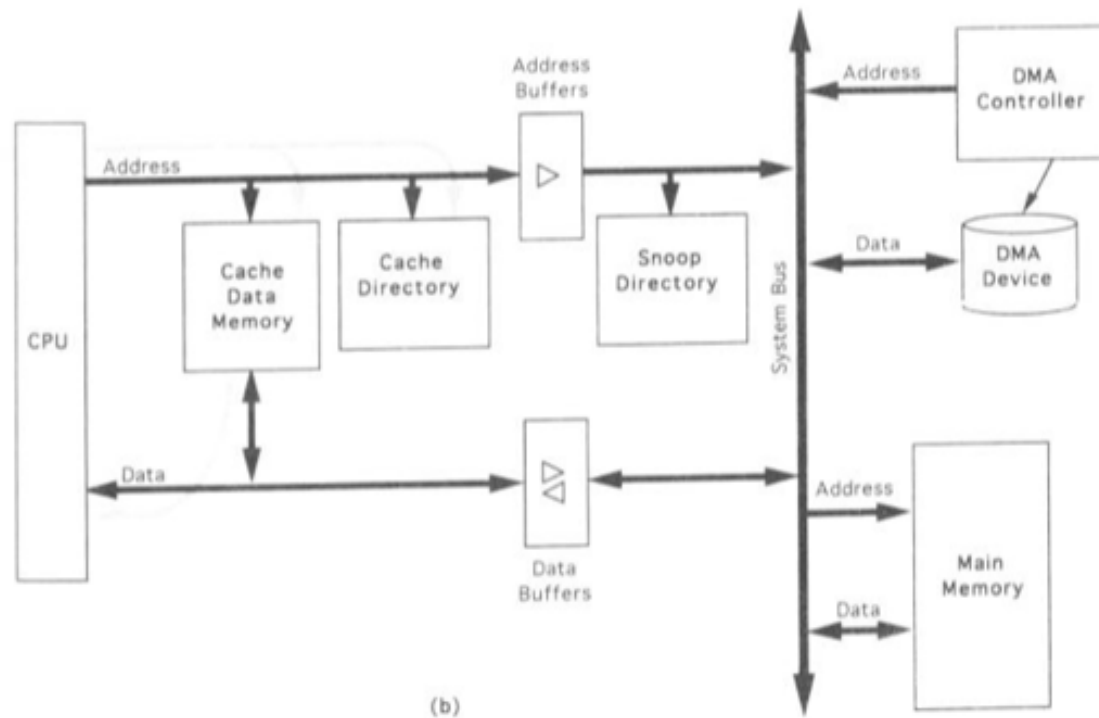
A one CPU, two level cache system:



A two CPU, one cache system:

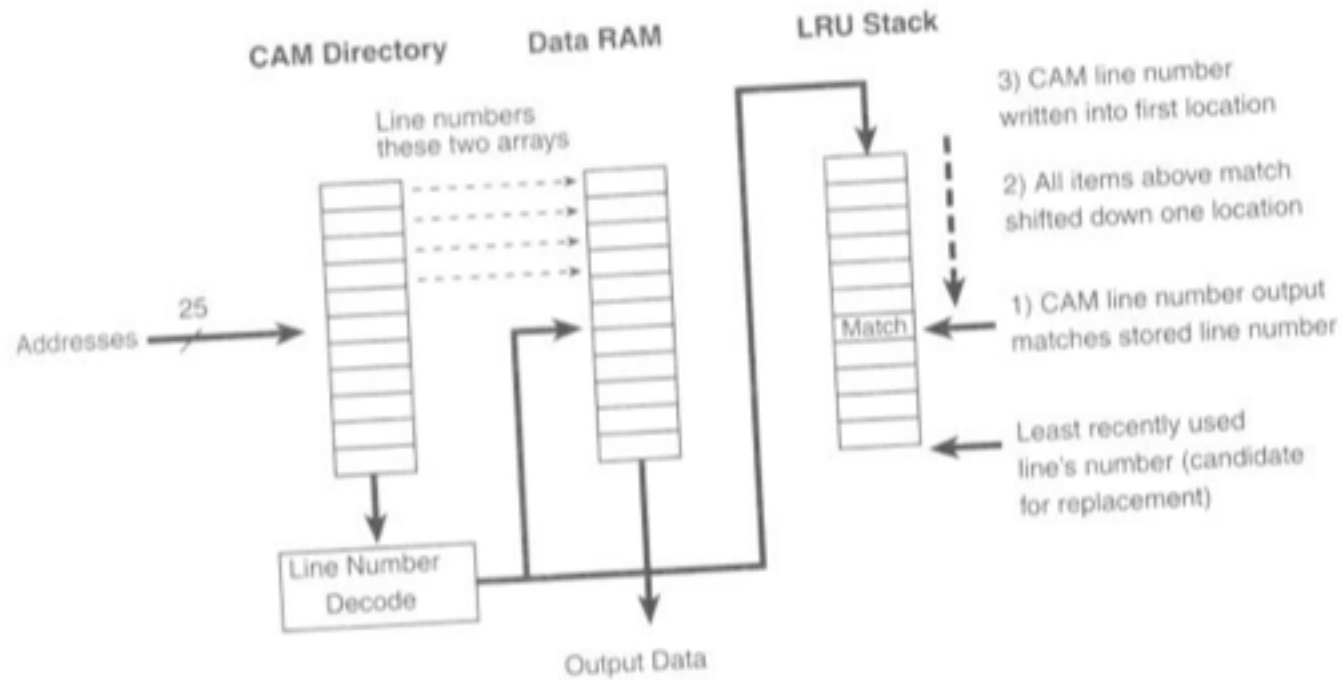


A one CPU, snooping cache system:



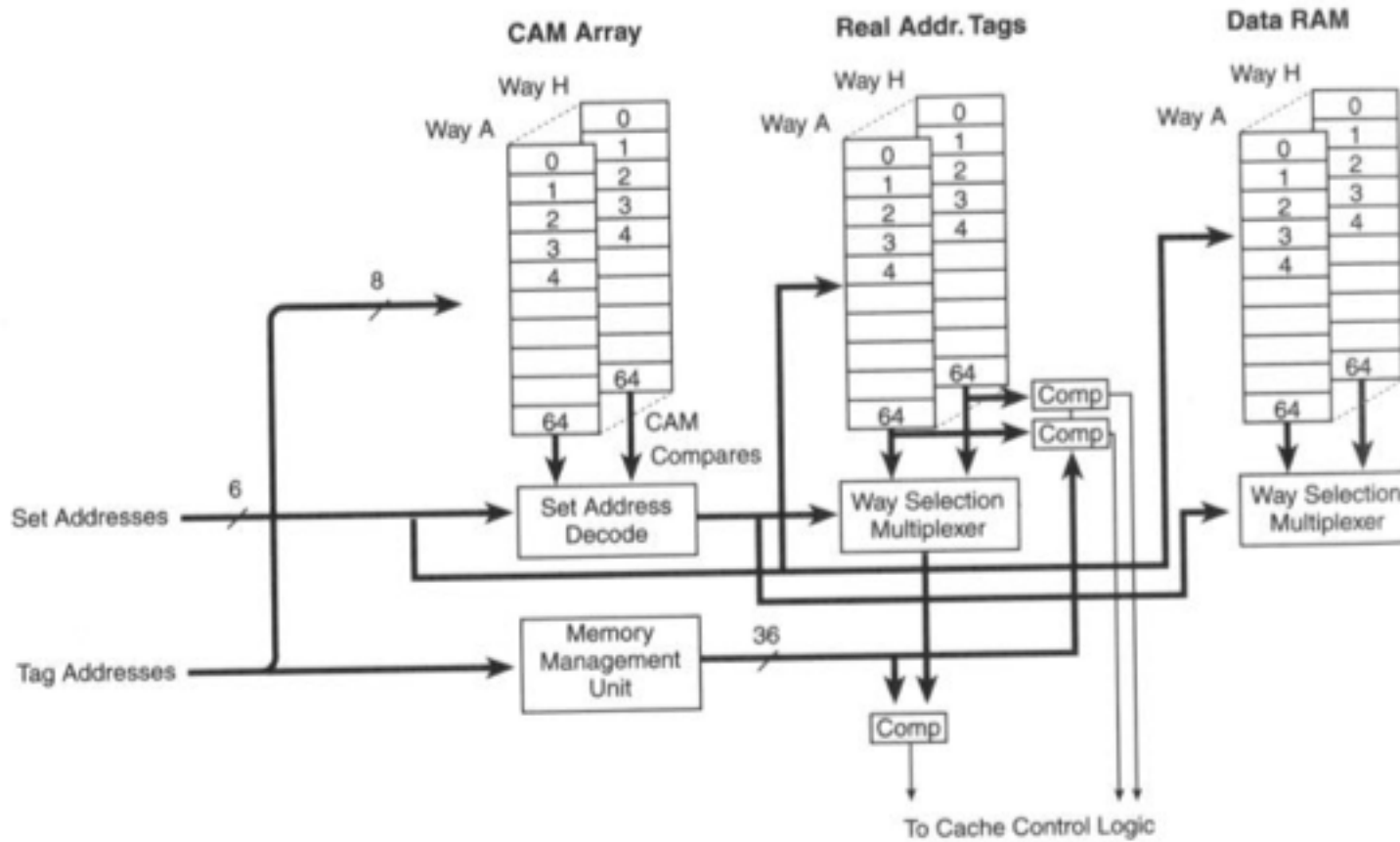
When a new cache line is added, the address is on both sides of the system bus so the cache and snoop directory are updated simultaneously. Otherwise, DMA/snooping + CPU/caching run.

True Least Recently Used (LRU) cache design:



IBM had ~2,000 people think about this design over several years.

IBM's semi-associative cache design:



Six basic classes of additional complications from modern CPU designs come from *instruction level parallelism* (ILP):

- Superscalar means a core can execute more than 1 operation per cycle:
 - 2-4 integer operations
 - 1-4 floating point operations (e.g., multiply-add pairings)
- Pipelined:
 - Floating point operations take $O(10)$ cycles to complete.
 - Start a new set of floating point operations every cycle if possible (vectorization).
- Multiple issue: Start a new set of operations every cycle
 - Static: functional units scheduled by compiler.
 - Dynamic: functional units scheduled at runtime.

- Load/store:
 - All operations are done on data in registers and all operands must be copied to/from memory using load and store operations, which may touch one or caches and main memory.
- Data alignment
- Speculation:
 - Branching decisions (if-then-else clauses)
 - Multiple issue usage: compiler makes an educated guess how to do multiple issue. Possibly multiple parts of a program are executed simultaneously and only one of the computations is used.

All of these “features” add up to needing a very good compiler and a lot of “hand” optimization, unfortunately.

Example of Pipelining and ILP:

```
for( int i = 0; i < n; i++ )  
    arrays[i][0] = arrays[i][1] + arrays[i][2];
```

Assume that $n = 10,000$ and addition takes 10 cycles. Then we get the last sum in $\sim 10,010$ cycles assuming that we do the indexing with instruction level parallelism with respect to the sums. This is much faster than the good old days when it would have taken $O(100,000)$ - $O(1,000,000)$ cycles (e.g., at least 400,000 cycles).

Multiple issue of instructions can be used to break the addition up into M pieces that are added up separately using separate hardware floating point units to provide even more parallelism on a single core. This complicates cache usage significantly.

Example of Speculation:

```
a = b + c;  
if ( a > 3.14 ) then  
    d = 2 * a;  
else  
    d = a + 42.0;
```

Speculation can either:

- Calculate just one of them and then calculate the other one if the guess is wrong (usual practice).
- Calculate both values of d and then store the right one.

Quick summary of how to find hotspots for cache tuning:

- Time sections of codes.
 - Coarse grained timing
 - Use the *time* function from the command line on a UNIX-like machine. Depending on the options you may be able to get the following information:
 - User time (secs.)
 - System time (secs.)
 - Wall clock time (aka total elapsed time)
 - Average amount of shared memory (program)
 - Average amount of unshared memory (data)
 - Number of page faults
 - Correlating the amount of unshared memory versus the number of page faults will help determine how well or badly data is being used in the program.

- Fine grained timing
 - Use a timer inside of program
 - Watch out for compiler optimizations that remove unreachable or unusable code.
 - Tends not to work well for short, fast loops.
 - Sometimes requires tricks to get a decent timing, e.g., put a function call with an array argument that just returns (and put the function in another file for compiling).
- Profiling of functions.
 - Tells you how much time you spend in a given function.
- Call graphs.
 - Tells you what percentage is spent in a tree of calls to functions and what the functions call.
 - Combine with profiling gives you hotspot information.

- Keep a clean machine.
 - Stop processes running on your computer that you do not need.
 - Most supercomputer nodes run “light weight” operating systems with minimal services running.
- Low hanging fruit syndrome.
 - Function A is the hotspot and uses 50% of the time. It is fixed so that it now uses 25% of the time.
 - Function B used to use 20% of the time and now uses a higher percentage. Fix it so that it uses less time, but the percentage reduction needs to be scaled with respect to the original percentage.
 - As you fix more functions, the actual speedup is less and less with respect to the original runtime.
 - What do you do with a 50,000 line hotspot???

- Amdahl's Bottleneck.
 - Loosely put, this states that execution time is dominated by the parts that cannot be improved.
 - In a parallel environment, this states that execution time is dominated by the serial parts of the application.
- Objects and data structures.
 - Well designed data structures should be used to implement algorithms and need to reflect the memory hierarchy to be efficient. Questions that arise are
 - Is the object statically or dynamically allocated?
 - Is the object globally or locally accessible?
 - If the object is dynamically allocated, is it allocated as one large chunk or in multiple small chunks and how often are the chunks allocated/freed/accessed?

- Is the data structure large or small and what unit of memory will it fit in (register, cache line, memory page, or bigger)?
- Inner loops
 - The innermost loop is particularly important to code in a cache aware style.
 - Analysis of how data is accessed in such a loop can provide a better definition of a data structure.
 - Analyze how the innermost loop interacts with the outer loop(s).
 - Analyze how different iterations of the innermost loop works.
- Hash tables
 - This is a basic data structure that caches and virtual memory is based on. Yet they are a disaster normally.

- Hash table collisions: frequently dealt with a linked list holding a <key,data> pair.
 - You have to walk through the linked list to find the correct key when doing a hash table lookup.
 - Linked lists frequently have very bad cache properties since malloc() is used and returns pointers all over virtual memory with no regard for physical memory.
- Step 1 is to build the hash table. Step 2 is to use the hash table.
 - Once the has table is built in Step 1, rebuild it all in one large chunk of memory before proceeding with Step 2.
 - Data locality will be the result.

How to make your code run fast on any machine

- Use the best algorithm(s)
- Use efficient libraries
- Find good compiler options
- Use suitable data layouts
- Use good runtime tools

Example: Solution of linear systems arising from the discretization of a special PDE

- Gaussian elimination (standard): $n^3/3$ ops
- Banded Gaussian elimination: $2n^2$ ops
- SOR method: $10n^{1.5}$ ops
- Multigrid method: $30n$ ops

Multigrid always wins when n is large enough. However, most problems have several methods with a similar complexity and the best implementation wins.

Efficient libraries

- There are preloaded libraries available on many HPC machines. Look for them in the web pages for a specific machine (e.g., Noor).
- The implementations use highly nonobvious, clever tricks. Do not expect to find the source code online necessarily.
- They may be written for and tuned for quite specific machines. Some are compiled with absolutely no optimization since it has all been done by hand.

Common HPC libraries:

- Basic linear algebra subroutines (BLAS) including ATLAS
- LAPACK and SCALAPACK/BLACS
- Vendor scientific libraries: IBM, Intel, HP, NAG, IMSL, ...
- Communications libraries for MPI, threading, shared memory

Compiler flags

- Read the manual and help information about your compiler.
- Cheat and use flags from the SPEC benchmarks (e.g., see <http://www.spec.org/benchmarks.html>).
- Ask your local supercomputer center/IT staff for recommendations.

Data layouts

Many languages follow declarations and do not optimize the data layout for you (e.g., C/C++/java/C#/...). Hence, you should declare data in the following order:

- Larger first followed by smaller ones.
- Arrays before scalars.

- Array size adjustments (padding):
 - On direct mapped cache systems, try to keep arrays declared consecutively to have different sizes (even if logically they should be the same).
 - On associative cache systems, keep arrays logically of a length divisible by the size of a cache line.
 - Similar tricks for data structures, too.
- Data (structures) should be aligned with cache lines.
- Elements of a data structure should not cross cache lines unintentionally (char data can be a real problem here).
- Place data variable that are accessed frequently together.
- Use static variables.
- Use malloc() to manage your own memory and verify alignment at runtime (consider vmalloc).

Runtime tools

- There are many interesting tools for getting information at runtime about your code, e.g.,
 - valgrind (memory leaks, cache usage, etc.)
 - valgrind program
 - cachegrind program
 - papi (<http://icl.cs.utk.edu/papi>)
 - TotalView
 - HPCToolkit (<http://hpctoolkit.org>)

Many of the best tools for gaining insight into performance run best on Linux or IBM Blue Gene/P. Some work on Windows. Mac OS/X is usually ignored.

Screen shot for HPCToolkit:

sweep3dsingle (02/25/02 11:45:43) - Netscape

File Edit View Go Communicator Help

Reset Restart **sweep3dsingle** Help

Source Files:

/home/rjf/HARD_C

decomp.f
driver.f
flux_err.f
initialize.f
inner.f
inner_auto.f
new_stuff.cpp
output.f
read_input.f
source.f
sweep.f
timers.c

Other Files:

_filbuf.c
_flsbuf.c
access.s
atexit.c
bcopy.s

SOURCE FILE: /home/rjf/HARD_CODES/sweep3d_alphas/sgiv1/sweep.f

```

516
517 c compute flux Pn moments (I-line)
518 c      original
519     do i = 1, it
520       flux(i,1,j,k) = flux(i,1,j,k) + w(m)*phi(i)
521     end do
522     do n = 2, nm
523       do i = 1, it
524         flux(i,n,j,k) = flux(i,n,j,k)
525           + pn(n,m,iq)*w(m)*phi(i)
526       end do
527     end do
528
529 c compute DSA face currents (I-line)
530 if (do_dsa) then
531   do i = 1, it
532     face(i+13,j,k,1) = face(i+13,j,k,1)
533       + wmu(m)*phi(i)
534   enddo

```

file pane

source pane

Location	sorted	Cycles	%	sort	L1 miss	%	sort	L2 miss	%	sort	TLB miss	%	sort	FP insts	%
Program	1.62e+10	100		5.63e+08	100		1.36e+07	100		7.43e+06	100		1.71e+09	100	
Ancestor	LP 215-646:sweep.f	1.33e+10	82	5.01e+0						+06	94		1.25e+09	73	
Current	LP 358-547:sweep.f	1.31e+10	81	4.93e+08	81					4			1.24e+09	72	
Descendants	LP 523-524:sweep.f	2.66e+09	16	1.11e+08	20		2.32e+06	17		5.49e+04	1		2.16e+08	13	
	LP 402-415:sweep.f	1.49e+09	9	8.26e+07	15		1.91e+06	14		1.38e+05	2		1.19e+08	7	
	sweep.f: 448	1.48e+09	9	4.64e+07	8		2.88e+						46e+08	14	
	sweep.f: 540	1.17e+09	7	4.02e+07	7		9.36e+						85e+07	5	
	sweep.f: 536	1.06e+09	7	3.86e+07	7		8.97e+						91e+07	4	
	sweep.f: 532	1.02e+09	6	3.83e+07	7		8.03e+05	6		9.19e+05	12		4.56e+07	3	
	sweep.f: 520	9.22e+08	6	3.74e+07	7		9.01e+05	7		1.36e+06	18		5.85e+07	3	
	sweep.f: 446	5.49e+08	3	4.06e+06	1		2.09e+05	2		1.31e+05	2		2.51e+07	1	
	sweep.f: 453	3.48e+08	2	4.93e+06	1		2.17e+05	2					9.08e+07	5	
	sweep.f: 401	3.22e+08	2	1.31e+07	2		1.07e+06	8		1.33e+06	18		2.42e+05	0	

parent scope

current scope

child scopes

flatten/
unflatten

Document: Done

HPCToolkit philosophy:

- Intuitive, top down user interface for performance analysis
 - Machine independent tools and GUI
 - Statistics to XML converters
 - Language independence
 - Need a good symbol locator at run time
 - Eliminate invasive instrumentation
 - Cross platform comparisons
- Provide information needed for analysis and tuning
 - Multilanguage applications
 - Multiple metrics
 - Must compare metrics which are causes versus effects (examples: misses, flops, loads, mispredicts, cycles, stall cycles, etc.)
 - Hide getting details from user as much as possible

- Eliminate manual labor from analyze, tune, run cycle
 - Collect multiple data automatically
 - Eliminate 90-10 rule
 - 90% of cycles in 10% of code... for a 50K line code, the hotspot is only 5,000 lines of code. How do you deal with a 5K hotspot???
 - Drive the process with simple scripts

Writing Optimizable Code

Many times, learning what *not to do* is as important as to learning what *to do*.

Excessive use of registers

- Using the *register* definition on too many scalar variables causes *register spillage*.
 - The compiler has to copy data from a register to memory to accommodate another register scalar.
 - Data goes all the way to main memory and back, which takes many cycles and is probably unnecessary. It also disrupts the cache in a bad way.
 - Let a compiler choose which scalars are register candidates through the optimizer.

Excessive use of globals

- Global variables are considered *evil* by optimizers.
- Any module can modify the value of a global variable so all values have to be written back to main memory. In a shared memory, multi-threaded environment, this is particularly troublesome.
- Adds to global optimization complexity and increases the chances for register spill situations.
- If globals are used when locals would suffice, compilers do not necessary recognize localness of actions that can normally be optimized heavily and well.
 - Loop variables are good examples. At the end of a loop the loop control variable must be written back to memory instead of possibly ignored.
 - Watch out for global pointers (more coming).

Functions and function calls

- Languages like C and C++ allow (encourage?) putting functions into separate files for compiling.
- Compilers do not usually do optimization across files.
- Within a file a compiler can analyze all of the functions and make optimizations.
- Most compilers assume that all function calls are destructive to the register and cache structure and stop certain types of optimizations.

Excessive use of pointers

- Pointers are poison to optimizers.
- Pointers can use overlapping memory inside of variables or data structures that a compiler cannot detect.
 - Runtime systems can detect the overlap.

- Only compilers that generate multiple sets of executable code for no overlap, full overlap, or partial overlap(s) and branch to proper code segment can produce fast, correct code. Not many compilers exist that do this, however.

```
void vector_copy( double* a, double* b, int n ) {  
    for( int i = 0; i < n; i++ )  
        a[i] = b[i];  
}
```

should be easy to optimize, but is not ☹

Not giving hints to the compiler

- *const* in C and C++ tells the compiler that something will not change in a function.
- *-noalias* (or equivalent) says no pointer overlaps in a file.

Self modifying code

- This is a really bad idea from a security viewpoint.
- It also kills the instruction cache and all instruction level parallelism that the hardware has detected.
- There can be a cache coherence problem on multicore systems.

That was a lot of negativity. So let's look at some positive things to do that are somewhat intrusive to a code. This is where efficient libraries are really useful: someone already did the nasty, intrusive, hard work for you already. You just have to learn how to call their routines and how to link their libraries. You may have to go on the Internet, download their library, and install it on your computer or in your account, however.

Loop unrolling

- Simplest effect of loop unrolling: fewer test/jump instructions (fatter loop body, less loop overhead).
- Fewer loads per Flop.
- May lead to threaded code that uses multiple FP units concurrently (instruction-level parallelism).
- How are loops handled that have a trip count that is not a multiple of the unrolling factor?
- Very long loops may not benefit from unrolling (instruction cache capacity).
- Very short loops may suffer from unrolling or benefit strongly.

Example: daxpy (from the BLAS)

```
double  a[n], b[n], c;  
int      i, ii;
```

```
for( i = 0; i < n; i++ )           // traditional form  
    a[i] = a[i] + c * b[i];
```

```
ii = n % 4;                        // 4-way unrolled form  
for( i = 0; i < ii; i++ )          // preliminary loop
```

```
    a[i] = a[i] + c * b[i];  
for( i = 1+ii; i < n; i += 4 ) {    // unrolled loop  
    a[i] = a[i] + c * b[i];  
    a[i+1] = a[i+1] + c * b[i+1];  
    a[i+2] = a[i+2] + c * b[i+2];  
    a[i+3] = a[i+3] + c * b[i+3];  
}
```

Loop unrolling can lead to better Flop/load ratios, e.g.,

```
for( i = 0; i < n; i++ )  
    for( j = 0; j < n; j++ )  
        y[i] = y[i] + a[i][j] * x[j];
```

has 3 Flops and 3 loads per iteration of the innermost loop, but

```
for( i = 0; i < n; i+=2 ) {  
    t1 = 0.0;  
    t2 = 0.0;  
    for( j = 0; j < n; j++ ) {  
        t1 = t1 + y[i] + a[i][j] * x[j] + a[i][j+1] * x[j+1];  
        t2 = t2 + y[i+1] + a[i+1][j] * x[j] + a[i+1][j+1] * x[j+1]; }  
    y[i] = t1;  
    y[i+1] = t2; }
```

has 8 Flops and 8 loads per iteration of the innermost loop.

Watch for register spills, however, if you go to 4-way loop unrolling!

Software pipelining

- Arranging instructions in groups that can be executed together in one cycle.
- Idea is to exploit instruction-level parallelism.
- Often done by optimizing compilers, but not always successfully
- Closely related to loop unrolling
- Less important on out-of-order execution based CPUs

Consider the daxpy example again. We can take the simple code and rewrite it into a real mess that no one would normally recognize. The simplest version appears on the next slide. Vendor versions of daxpy follow this example, but optimize the loop unrolling to an extreme.

```
t2 = a[0];
t1 = b[0];
r = t1 * c;
t3 = t2;
t2 = a[1];
t1 = b[1];
for( i = 2; i < n-1; i++ ) {
    a[i-2] = r + t3;
    r = t1 * c;
    t3 = t2;
    t2 = a[i];
    t1 = b[i]; }
a[n-2] = r + t3;
r = t1 * c;
t3 = t2;
t2 = a[n-1];
t1 = b[n-1];
a[n-1] = r + t3;
```

Special functions

/ (divide), sqrt, exp, log, sin, cos, ...

are expensive, even if done in hardware using fancy algorithms with built-in data lookup.

- Use math Identities, e.g., $\log(x) + \log(y) = \log(x*y)$.
- Use special libraries that
 - vectorize when many of the same functions must be evaluated.
 - trade accuracy for speed, when appropriate.

if statements ...

- Prohibit some optimizations (e.g., loop unrolling in some cases).
- Evaluating the condition expression takes time.
- CPU pipeline may be interrupted.
- Dynamic jump prediction issues.

Goal: avoid if statements in the innermost loops

No generally applicable technique exists ☹

Function calling overhead

- Functions are very important for structured, modular programming.
- Function calls are expensive, $O(100)$ cycles.
- Passing value arguments (copying data) can be extremely expensive, when used inappropriately.
- Passing reference arguments may be dangerous from a point of view of correct software.
- Reference arguments (as in C++) with *const* declaration.
- Generally, in tight loops, no subroutine calls should be used.
- *Inlining* (C++) can be quite useful, whether automatically by the compiler or by hand.
- Macros can lead to lots of problems, e.g.,
`#define squre(a) a*a`

What can go wrong? Well...

`squire(x+y)` maps to $x+y * x+y$ (oops)

`squire(f(x))` maps to $f(x) * f(x)$

Depending on the definition of $f()$, two calls might return different results for the same input. Even if there are no side effects, the compiler might not deduce this fact and make bad decisions concerning the instruction pipeline and cache state.

Loop fusion

- Transform successive loops into a single loop to enhance temporal locality.
- Reduces cache misses and enhances cache reuse (exploit temporal locality).
- Often applicable when data sets are processed repeatedly.

Example:

```
for( i = 0; i < n; i++ )           // a loads into cache twice if it
    a[i] = a[i] + b[i];           // is big enough.
for( i = 0; i < n; i++ )
    a[i] = a[i] * c[i];
```

but

```
for( i = 0; i < n; i++ )           // a loads into cache exactly once
    a[i] = (a[i] + b[i]) * c[i];
```

Loop splitting (loop fission) is the inverse of loop fusion. A loop is split into at least two loops to leverage compiler optimizations and instruction cache effects. This can aid multithreading, too.

Loop interchanging can occur when the inner and outer loops do not change the computed results inside the loops. The exchange occurs when the inner loop has fewer iterations than the outer loop and caching is not effected by the interchange. Data dependencies have to be analyzed carefully before a loop interchange.

Loop reversal occurs when the order of the loop is invariant to the results. This is usually combined with loop fusion or fission.

Loop blocking (loop tiling or strip mining) can provide big improvements. Data dependency analysis is crucial here.

- Divide the data set into subsets (blocks) that are small enough to fit in cache.
- Perform as much work as possible on the data in cache before moving to the next block.

Example: matrix-matrix multiplication with a block size of s

```
for( i = 0; i < n; i++ )
    for( j = 0; j < n; j++ )
        for( k = 0; k < n; k++ )
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

becomes

```
for( ii = 0; ii < n; ii += s )  
    for( jj = 0; jj < n; jj += s )  
        for( kk = 0; kk < n; kk += s )  
            for( i = ii; i < ii+s; i++ )  
                for( j = jj; j < jj+s; j++ )  
                    for( k = kk; k < kk+s; k++ )  
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Distributed Parallel Computing

What's the problem?

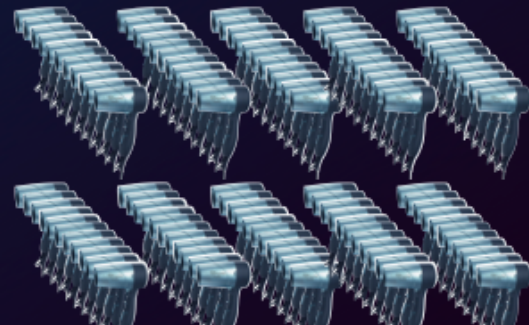
with four strong jet engines
(not those of Rolls-Royce of course)

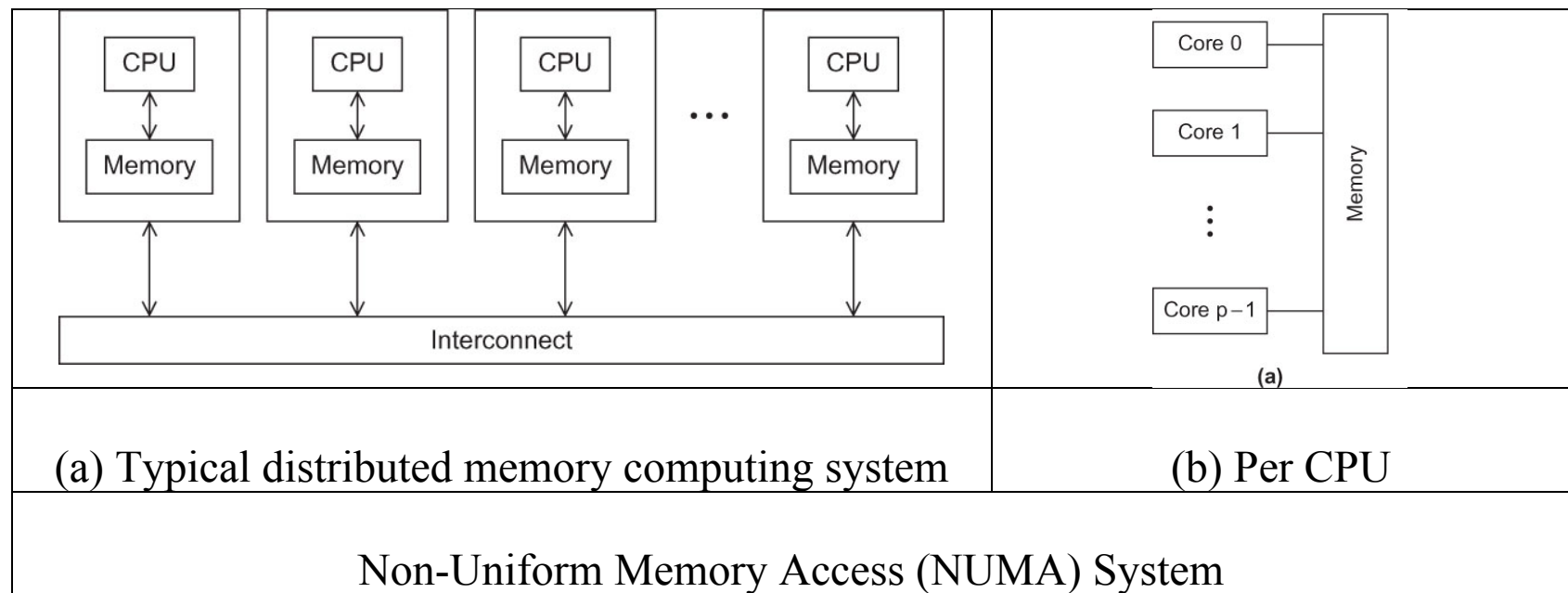


Would you want to propel
a Super Jumbo



or with 300,000
blow dryer fans?





How many memory levels are hiding in this diagram?

1. Main memory
2. Local shared memory
3. Cache memory (2-3 caches)
4. Registers

So, 5-6 levels!

Flynn's Taxonomy and What Is Used Today

M. J. Flynn, Some Computer Organizations and Their Effectiveness, *IEEE Transactions on Computing* C-21, No. 9, Sep 1972, pp .948-960.

- **SISD** (single instruction, single data): von Neumann model.
- **SIMD** (single instruction, multiple data): apply same instruction to multiple data streams.
- **MISD** (multiple instruction, single data): not too many exist.
- **MIMD** (multiple instruction, multiple data): all processes do whatever they want with multiple data streams.

SPMD (single program, multiple data): all processes run the same program on multiple data streams and each process does whatever it wants using conditional branches (F. Darema et al, 1988).

What is the golden system today?

- **MPI** (message passing interface): many MPI implementations
- MPI+OpenMP
- MPI+OpenMP+{CUDA, OpenCL, ... (gp-gpu interfaces)}

What is the golden system by the time an Exascale system works?

- Nobody knows, but it probably will not be any of the ones from today.
- There is a feeling that now is the time to review all aspects of communication and parallel programming and to invent a better system.
 - Sounds good on paper and in talks.
 - Much harder to get anyone to actually use, e.g., consider Fortran's non-demise over the past 20+ years of certain, imminent death predictions.

Issues:

- Embarrassingly parallel: almost no communication and perfect scaling. Just what everyone wishes for in a parallel algorithm.
- Load balanced: Every processor runs for the same amount of time.
- Combining the two:
 - Usually not possible (data dependencies, algorithms, etc).
 - Sometimes possible with a NP-complete algorithm that is too slow in the worst case to be practical.
 - Papers have been produced with good approximations to optimal scaling and parallelization for many problems.
 - How to divide an unstructured mesh for solving a PDE with a finite element method that minimizes data communication between processors? (literally, 100's of proposed solutions)

What does MPI look like (C):

- Has a main program
- Has include files
 - `stdio.h`, `stdlib.h` `string.h`, ...
 - Have to add `mpi.h`
 - Functions have form `MPI_Xyzzy`
 - MPI all capital letters
 - Then Capital+all lower case letter and `_`'s.
 - `MPI_Send()` or `MPI_Isend()`
 - Constants have form `MPI_XYZZY` (all capital letters)

Compiling and linking MPI programs:

- Use `mpicc` (C) or `mpic++` or `mpicxx` (C++).
- Do not use regular C/C++ compiler commands.

Running a MPI program

- Unless noted, use either `mpirun` or `mpiexec`
- On Noor, use `mpirun.lsf`
 - Follow the instructions in the web page <http://rcweb.kaust.edu.sa/KAUST/ResearchComputing/wiki/NoorGuide> including how to load the correct versions of MPI, OpenMP, CUDA, etc. (somewhat painful at first until you are used to the system).

MPI is

- Primarily a communications library.
- It is extremely low level, akin to assembly language.
- It is designed for simple users and compiler writers.
- The number of parallel processes is set to a fixed maximum. It is not dynamic. *Process failures are problematic.*

Example: Hello world in C

```
#include <stdio.h>
```

```
#include <mpi.h>
```

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

```
    int rank, size;
```

```
    MPI_Init (&argc, &argv);    /* starts MPI */
```

```
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);    /* get current process id */
```

```
    MPI_Comm_size (MPI_COMM_WORLD, &size);    /* get number of processes */
```

```
    printf( "Hello world from process %d of %d\n", rank, size );
```

```
    MPI_Finalize();
```

```
    return 0;
```

```
}
```

Communicators in MPI

`MPI_COMM_WORLD` represents all of the processes that started with the `mpirun/mpiexec` command.

Other communicators can be created that use a subset of the processes. Quite complicated programs may use multiple communicators to create the equivalent of `SECTIONS` from OpenMP.

Within a communicator of size *rank*, each process is numbered from 0 to *rank-1*.

Process 0 frequently does different things than the rest (SPMD model). Take care what you do in process 0.

Scaling: Speedup and Efficiency

Speedup is defined as the ratio of the *fastest* serial code to a parallel code:

$$S(n,p) = \frac{T_{serial}(n)}{T_{parallel}(n,p)}.$$

Often $S(n,p)$ uses $p=1$ in a parallel code for the numerator instead of a faster serial algorithm in a different code. This is misleading at best and outright cheating at worst.

Determining what to use for the numerator is frequently quite challenging if honest speedups are presented.

Efficiency can also be a challenge. Consider

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{serial}(n)}{p \times T_{parallel}(n,p)}.$$

In this case using $p=1$ in the parallel code is usually the correct choice since this is a measure of how well the parallel code scales as more processors are used. However, $p=1$ for massively parallel codes with tremendous amounts of data is not available since a single node does not have enough memory to run the case. Hence,

$$E(n,p,p_0) = \frac{p_0 \times T_{parallel}(n,p_0)}{p \times T_{parallel}(n,p)}, p > p_0.$$

Example: Ax , square A of order n , p cores

Speedup	n				
p	1024	2048	4096	8192	16384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Efficiency	n				
p	1024	2048	4096	8192	16384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.98	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

A program is

- **Scalable** if the problem size can be increased at a rate so that the efficiency does not decrease as the number of processors increase.
- **Strongly scalable** if a constant efficiency is maintained without increasing the data size.
- **Weakly scalable** if a constant efficiency is maintained only by increasing the data size at the same rate as the processor increase.

Many scalable applications are strongly scalable up to some number of processors and a subset is then weakly scalable thereafter.

Most applications are only scalable to a point.

Parallel computing models

A lot can be learned before writing a single line of code by analyzing how much time communications will take with respect to computing time.

Communications time is measured by message length, bandwidth, and latency time. To move a message of N bytes, it takes

$$T(N, p1, p2) = L_{p1, p2} + N/B_{p1, p2}, \text{ where}$$

- $L_{p1, p2}$ is the latency time to get the first byte(s) between $p1$ and $p2$
- $B_{p1, p2}$ is the time to move a byte in a sustained manner between $p1$ and $p2$.

Modeling computational time used to be easy: just count the number of floating point and possibly the integer operations and scale it by time per operation.

Now memory subsystems are important, too.

Modeling by cache misses would be extremely valuable, but is not practical due to the randomness of how this operates too often.

It is better to blend the arithmetic model with a model of how it takes on average to move blocks of data between main memory and the CPU.

Performance ratings

Which is more important?

- Timing
- FLOPS (floating point ops/sec)

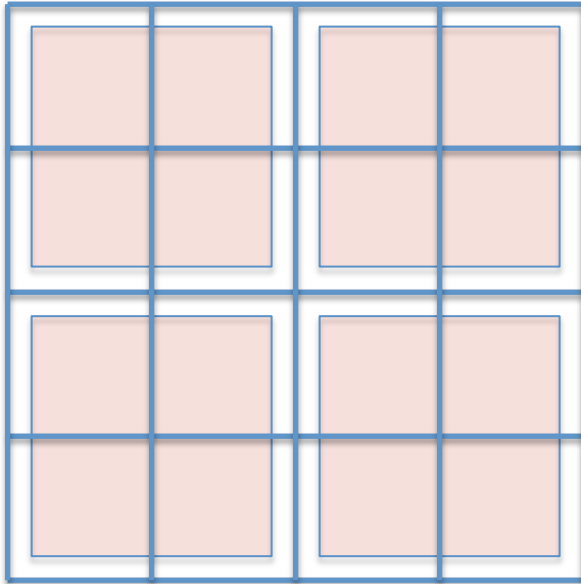
It depends on who is doing the ranking:

- TOP500.org and IT directors: FLOPS
- Real people who need a solution: Timing

I would rather be known as really, really fast instead of producing lots and lots of flops*.

* In English, a flop is synonymous with a failure.

Example of scaling: Gauss-Seidel on a mesh (intersection of lines)



For any (i,j) in the mesh, compute

$$u_{ij} = (a_{ii})^{-1} (f_{ij} - a_{i,j-1}u_{i,j-1} - a_{i,j+1}u_{i,j+1} - a_{i-1,j}u_{i-1,j} - a_{i+1,j}u_{i+1,j}),$$

$i=1, \dots, n-2$ and $j=1, \dots, n-2$

No obvious parallelism!

First, divide domain up into subdomains (light red areas).

- Gauss-Seidel within subdomains is not the same algorithm exactly: the order of the “smoothing” is different.
- Hence, u_{ij} are different than in the serial case.

Second, use “red-black” ordering: color the points, alternating.

- Compute just on the red points first.
 - Obvious 2-way parallelism, but actually much more possible.
 - Exchange data between neighboring subdomains after whole subdomains or individual lines.
 - Overlap communications and computing entirely if done right.
- Now compute just on the black points similarly.
- Much more parallelism, but convergence rate still not the same as the original serial algorithm.

What about asynchronous parallel algorithms?

- Just compute, communicate, and compute again.
- Use the latest data from neighboring subdomains.
- Does it converge to the solution of $AX=f$?
 - Not necessarily. ☹
 - Need extra conditions to avoid race conditions. ☺

A simple summary of distributed parallel computing

- Programming paradigms:
 - Many programs on traditional CPU based parallel clusters use SPMD style programming.
 - SIMD is useful on GP-GPU type architectures.
- Determinism:
 - Most serial programs are deterministic.
 - Most parallel programs are not deterministic.
- Are results identical?
 - When floating point numbers are in the solution, do not expect to get exactly the same result in parallel since roundoff errors will be different if the numbers are manipulated in a different order. Even if mathematically it should not make a difference, computationally it does.
 - Integer, character, ... *usually* the same.

- Timing:
 - Use wall clock times, not CPU times.
 - Total CPU time on a parallel computer is greater than the serial computer CPU time (remember, it is the sum of CPU times).
 - People want the results sooner, so a wall clock is watched.
- Scalability:
 - Speedup is important since it tells you how efficiently you are using added processors.
 - Different types of scalability are important to know for a given problem so as to request resources responsibly.
 - Parallel models are not often used, but should be since it provides a theoretical tool to measure practical implementations by to see if scaling occurs as predicted and expected.
 - Use the correct performance rating system even if FLOPS is the given measure.

Fast Libraries

Basic Linear Algebra Subroutines (*BLAS*)

- <http://www.netlib.org/blas>
- Level 1: vector-vector operations
- Level 2: matrix-vector operations
- Level 3: matrix-matrix operations
- Extended precision BLAS
- Sparse BLAS
- Parallel ... BLAS
- Specialized BLAS for C++, Java, Python in addition to the original ones written in Fortran, easily callable by Fortran/C
- Self tuning BLAS (ATLAS)

References

C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. on Math. Soft.* 5 (1979) 308-325

J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* 14,1 (1988) 1-32

J.J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* (1989)

Meaning of prefixes

S - REAL	C - COMPLEX
D - DOUBLE PRECISION	Z - COMPLEX*16
	(this may not be supported by all machines)

For the Level 2 BLAS a set of extended-precision routines with the prefixes ES, ED, EC, EZ may also be available.

Level 1 BLAS

	dim	scalar	vector	vector	scalars	5-element array
SUBROUTINE <code>zROTG</code> (A, B, C, S)	
SUBROUTINE <code>zROTMG</code> (D1, D2, A, B,	PARAM)
SUBROUTINE <code>zROT</code> (N,			I, INCX, Y, INCY,		C, S)	
SUBROUTINE <code>zROTH</code> (N,			I, INCX, Y, INCY,			PARAM)
SUBROUTINE <code>zSWAP</code> (N,			I, INCX, Y, INCY)			
SUBROUTINE <code>zSCAL</code> (N,	ALPHA,		I, INCX)			
SUBROUTINE <code>zCOPY</code> (N,			I, INCX, Y, INCY)			
SUBROUTINE <code>zAXPY</code> (N,	ALPHA,		I, INCX, Y, INCY)			
FUNCTION <code>zDOT</code> (N,			I, INCX, Y, INCY)			
FUNCTION <code>zDOTU</code> (N,			I, INCX, Y, INCY)			
FUNCTION <code>zDOTC</code> (N,			I, INCX, Y, INCY)			
FUNCTION <code>zxDOT</code> (N,			I, INCX, Y, INCY)			
FUNCTION <code>zNRM2</code> (N,			I, INCX)			
FUNCTION <code>zASUM</code> (N,			I, INCX)			
FUNCTION <code>IzAMAX</code> (N,			I, INCX)			

Generate plane rotation
 Generate modified plane rotation
 Apply plane rotation
 Apply modified plane rotation
 $x \leftrightarrow y$
 $x \leftarrow \alpha x$
 $y \leftarrow x$
 $y \leftarrow \alpha x + y$
 $dot \leftarrow x^T y$
 $dot \leftarrow x^T y$
 $dot \leftarrow x^H y$
 $dot \leftarrow \alpha + x^T y$
 $nrm2 \leftarrow \|x\|_2$
 $asum \leftarrow \|re(x)\|_1 + \|im(x)\|_1$
 $amax \leftarrow 1^{st} k \ni |re(x_k)| + |im(x_k)|$
 $\quad = \max(|re(x_i)| + |im(x_i)|)$

Level 2 BLAS

Level 1 BLAS

In addition to the listed routines there are two further extended-precision dot product routines DQDOTI and DQDOTA.

Level 2 and Level 3 BLAS

Matrix types:

GE - GEneral	GB - General Band	
SY - SYmmetric	SB - Sym. Band	SP - Sum. Packed
HE - HERmitian	HB - Herm. Band	HP - Herm. Packed
TR - TRiangular	TB - Triang. Band	TP - Triang. Packed

Level 2 and Level 3 BLAS Options

Dummy options arguments are declared as CHARACTER*1 and may be passed as character strings.

TRANx	= 'No transpose', 'Transpose', 'Conjugate transpose' (X, X^T, X^H)
UPLO	= 'Upper triangular', 'Lower triangular'
DIAG	= 'Non-unit triangular', 'Unit triangular'
SIDE	= 'Left', 'Right' (A or op(A) on the left, or A or op(A) on the right)

For real matrices, TRANSx = 'T' and TRANSx = 'C' have the same meaning.

For Hermitian matrices, TRANSx = 'T' is not allowed.

For complex symmetric matrices, TRANSx = 'H' is not allowed.

Level 2 BLAS

options	dim	b-width	scalar	matrix	vector	scalar	vector
xGEMV (TRANS,)	M, N,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xHEMV (TRANS,)	M, N, KL, KU,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xHEMV (UPLO,)	N,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xHEMV (UPLO,)	N, K,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xHPMV (UPLO,)	N,		ALPHA, AP, X, INCX, BETA, Y, INCY)				
xSYMV (UPLO,)	N,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xSBMV (UPLO,)	N, K,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xSPMV (UPLO,)	N,		ALPHA, AP, X, INCX, BETA, Y, INCY)				
xTRMV (UPLO, TRANS, DIAG,)	N,		A, LDA, X, INCX)				
xTBMV (UPLO, TRANS, DIAG,)	N, K,		A, LDA, X, INCX)				
xTPMV (UPLO, TRANS, DIAG,)	N,		AP, X, INCX)				
xTRSV (UPLO, TRANS, DIAG,)	N,		A, LDA, X, INCX)				
xTBSV (UPLO, TRANS, DIAG,)	N, K,		A, LDA, X, INCX)				
xTPSV (UPLO, TRANS, DIAG,)	N,		AP, X, INCX)				
options	dim	scalar	vector	vector	matrix		
xGER ()	M, N,	ALPHA, X, INCX, Y, INCY, A, LDA)					
xGERU ()	M, N,	ALPHA, X, INCX, Y, INCY, A, LDA)					
xGERC ()	M, N,	ALPHA, X, INCX, Y, INCY, A, LDA)					
xHER (UPLO,)	N,	ALPHA, X, INCX, A, LDA)					
xHPR (UPLO,)	N,	ALPHA, X, INCX, AP)					
xHER2 (UPLO,)	N,	ALPHA, X, INCX, Y, INCY, A, LDA)					
xHPR2 (UPLO,)	N,	ALPHA, X, INCX, Y, INCY, AP)					
xSYR (UPLO,)	N,	ALPHA, X, INCX, A, LDA)					
xSPR (UPLO,)	N,	ALPHA, X, INCX, AP)					
xSYR2 (UPLO,)	N,	ALPHA, X, INCX, Y, INCY, A, LDA)					
xSPR2 (UPLO,)	N,	ALPHA, X, INCX, Y, INCY, AP)					

— CONTINUED ON NEXT PAGE —

$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$
 $y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$
 $y \leftarrow \alpha Ax + \beta y$
 $y \leftarrow \alpha Ax + \beta y$
 $y \leftarrow \alpha Ax + \beta y$
 $y \leftarrow \alpha Ax + \beta y$
 $y \leftarrow \alpha Ax + \beta y$
 $y \leftarrow \alpha Ax + \beta y$
 $x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$
 $x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$
 $x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$
 $x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$
 $x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$
 $x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$
 $A \leftarrow \alpha xy^T + A, A - m \times n$
 $A \leftarrow \alpha xy^T + A, A - m \times n$
 $A \leftarrow \alpha xy^H + A, A - m \times n$
 $A \leftarrow \alpha xx^H + A$
 $A \leftarrow \alpha xx^H + A$
 $A \leftarrow \alpha xy^H + y(\alpha x)^H + A$
 $A \leftarrow \alpha xy^H + y(\alpha x)^H + A$
 $A \leftarrow \alpha xx^T + A$
 $A \leftarrow \alpha xx^T + A$
 $A \leftarrow \alpha xy^T + \alpha yx^T + A$
 $A \leftarrow \alpha xy^T + \alpha yx^T + A$

Level 3 BLAS

options	dim	scalar matrix	matrix	scalar matrix	
xGEMM (TRANS, TRANSB,	M, N, K,	ALPHA, A, LDA, B, LDB, BETA, C, LDC)			$C \leftarrow \alpha \text{op}(A)\text{op}(B) + \beta C, \text{op}(X) = X, X^T, X^H, C - m \times n$
xSYMM (SIDE, UPLO,	M, N,	ALPHA, A, LDA, B, LDB, BETA, C, LDC)			$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$
xHEMM (SIDE, UPLO,	M, N,	ALPHA, A, LDA, B, LDB, BETA, C, LDC)			$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$
xSYRK (UPLO, TRANS,	N, K,	ALPHA, A, LDA, BETA, C, LDC)			$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$
xHERK (UPLO, TRANS,	N, K,	ALPHA, A, LDA, BETA, C, LDC)			$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$
xSYR2K(UPLO, TRANS,	N, K,	ALPHA, A, LDA, B, LDB, BETA, C, LDC)			$C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C, C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C, C - n \times n$
xHER2K(UPLO, TRANS,	N, K,	ALPHA, A, LDA, B, LDB, BETA, C, LDC)			$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C, C - n \times n$
xTRMM (SIDE, UPLO, TRANS,	DIAG, M, N,	ALPHA, A, LDA, B, LDB)			$B \leftarrow \alpha \text{op}(A)B, B \leftarrow \alpha B\text{op}(A), \text{op}(A) = A, A^T, A^H, B - m \times n$
xTRSM (SIDE, UPLO, TRANS,	DIAG, M, N,	ALPHA, A, LDA, B, LDB)			$B \leftarrow \alpha \text{op}(A^{-1})B, B \leftarrow \alpha B\text{op}(A^{-1}), \text{op}(A) = A, A^T, A^H, B - m \times n$

2

This list is incomplete for doing HPC linear algebra. For example, it is missing a Strassen style matrix-matrix multiplication routine, which is much faster than xGEMM in the BLAS. Consider

<http://www.mgnet.org/~douglas/Codes/gemmww.tgz>,

C.C. Douglas, M.A. Heroux, G. Slishman, and R.M. Smith. GEMMW: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm. *Journal of Computational Physics*, **110** (1994), pp. 1-10.

LAPACK (replaces LINPACK and EISPACK)

- LAPACK can solve systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. LAPACK also handles many associated computations, e.g., matrix factorizations and estimating condition numbers.
- Dense and band matrices are provided for, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices.
- LAPACK uses the BLAS whenever practical. In some cases, there is no BLAS, so LAPACK extends the BLAS.
- All routines in LAPACK are provided in both single and double precision versions (real and complex data).

Structure of routines:

- **driver** routines, each of which solves a complete problem.
- **computational** routines, each of which performs a distinct computational task.
- **auxiliary** routines:
 - routines that perform subtasks of block algorithms
 - routines that perform some commonly required low-level computations
 - a few extensions to the BLAS

Normally, you use the drivers unless you really know what you are doing. There are simple and expert drivers for each operation. Expert drivers include extra operations like using the transpose of a matrix. When possible, use the simple drivers.

All driver and computational routines have names of the form `XYZZZZ` or `XYZZ`.

The first letter, X, indicates the data type as follows:

S	REAL
D	DOUBLE PRECISION
C	COMPLEX
Z	COMPLEX*16 or DOUBLE COMPLEX

When we wish to refer to an LAPACK routine generically, regardless of data type, we replace the first letter by x. Thus, `xGESV` refers to any or all of the routines `SGESV`, `CGESV`, `DGESV` and `ZGESV`.

The next two letters, **YY**, indicate the type of matrix (or of the most significant matrix). Most of these two-letter codes apply to both real and complex matrices; a few apply specifically to one or the other.

BD	bidagonal
DI	diagonal
GB	general band
GE	general (i.e., unsymmetric, in some cases rectangular)
GG	general matrices, generalized problem (i.e., a pair of general matrices)
GT	general tridiagonal
HB	(complex) Hermitian band
HE	(complex) Hermitian

HG	upper Hessenberg matrix, generalized problem (i.e., a Hessenberg and a triangular matrix)
HP	(complex) Hermitian, packed storage
HS	upper Hessenberg
OP	(real) orthogonal, packed storage
OR	(real) orthogonal
PB	symmetric or Hermitian positive definite band
PO	symmetric or Hermitian positive definite
PP	symmetric or Hermitian positive definite, packed storage
PT	symmetric or Hermitian positive definite tridiagonal
SB	(real) symmetric band
SP	symmetric, packed storage
ST	(real) symmetric tridiagonal
SY	symmetric
TB	triangular band

TG	triangular matrices, generalized problem (i.e., a pair of triangular matrices)
TP	triangular, packed storage
TR	triangular (or in some cases quasi-triangular)
TZ	trapezoidal
UN	(complex) unitary
UP	(complex) unitary, packed storage

The end letters **ZZ** and **ZZZ** indicate the computation performed.

The list of drivers (simple and expert versions) is quite long and should be looked up in the LAPACK Users Guide (either the SIAM book or online at <http://www.netlib.org/lapack/lug>).

Linear Equations

Two types of driver routines are provided for solving systems of linear equations:

- A **simple** driver (name ending -SV), which solves the system $AX = B$ by factorizing A and overwriting B with the solution X ;
- An **expert** driver (name ending -SVX) that can also perform the following (possibly optional) functions:
 - Solve $A^T X = B$ or $A^H X = B$ (unless A is symmetric or Hermitian);
 - Estimate the condition number of A , check for near-singularity, and check for pivot growth;
 - Refine the solution and compute forward and backward error bounds;
 - Equilibrate the system if A is poorly scaled.

Each expert driver requires roughly twice as much storage as the simple driver in order to perform these extra functions. Both types of driver routines can handle multiple right hand sides (the columns of B).

There are a lot of drivers for solving linear equations due to the variety of matrix storage types supported.

The total number is... (gasp!)

Type of matrix	Operation	Single precision		Double precision	
and storage scheme		real	complex	real	complex
general	simple driver	SGESV	CGESV	DGESV	ZGESV
	expert driver	SGESVX	CGESVX	DGESVX	ZGESVX
general band	simple driver	SGBSV	CGBSV	DGBSV	ZGBSV
	expert driver	SGBSVX	CGBSVX	DGBSVX	ZGBSVX
general tridiagonal	simple driver	SGTSV	CGTSV	DGTSV	ZGTSV
	expert driver	SGTSVX	CGTSVX	DGTSVX	ZGTSVX
symmetric/Hermitian	simple driver	SPOSV	CPOSV	DPOSV	ZPOSV
positive definite	expert driver	SPOSVX	CPOSVX	DPOSVX	ZPOSVX
symmetric/Hermitian	simple driver	SPPSV	CPPSV	DPSPV	ZPPSV
positive definite (packed storage)	expert driver	SPPSVX	CPPSVX	DPSPVX	ZPPSVX
symmetric/Hermitian	simple driver	SPBSV	CPBSV	DPBSV	ZPBSV
positive definite band	expert driver	SPBSVX	CPBSVX	DPBSVX	ZPBSVX
symmetric/Hermitian	simple driver	SPTSV	CPTSV	DPTSV	ZPTSV
positive definite tridiagonal	expert driver	SPTSVX	CPTSVX	DPTSVX	ZPTSVX
symmetric/Hermitian	simple driver	SSYSV	CHESV	DSYSV	ZHESV
indefinite	expert driver	SSYSVX	CHESVX	DSYSVX	ZHESVX
complex symmetric	simple driver		CSYSV		ZSYSV
	expert driver		CSYSVX		ZSYSVX
symmetric/Hermitian	simple driver	SSPSV	CHPSV	DSPSV	ZHPSV
indefinite (packed storage)	expert driver	SSPSVX	CHPSVX	DSPSVX	ZHPSVX
complex symmetric	simple driver		CSPSV		ZSPSV
(packed storage)	expert driver		CSPSVX		ZSPSVX

$$11 \times 2 \times 4 - 8 = 80!$$

Different methods of fast coding used in Lapack:

- Vectorization
 - Linear algebra vectorizes thanks to multiply-add pairings. However, many compilers make too many memory references, leading to slower than expected performance.
- Data movement impedes performance
 - Moving data between (vector) registers, different caches, and main memory all slow down performance. Re-use of data immediately (spatial locality principle) is imperative.
- Parallelism
 - On shared memory computers, nodes, or CPUs there is a lot of potential for parallelism in many linear algebra algorithms. On multicore CPUs this is a natural to exploit.
 - Most libraries are not thread enabled by default due to the age of the libraries. (Time for a rewrite?)

Example: Computing $A = U^T U$ (the Cholesky factorization), where A is symmetric, positive definite, $N \times N$, and U is upper triangular.

This can be done with Level 2 or 3 BLAS routines instead of writing the obvious code based on algorithms found in most numerical analysis books containing a section on numerical linear algebra.

Consider the core operation starting from

$$\begin{pmatrix} A_{11} & a_j & A_{13} \\ \cdot & a_{jj} & \alpha_j^T \\ \cdot & \cdot & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ u_j^T & u_{jj} & 0 \\ U_{13}^T & \mu_j & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & u_j & U_{13} \\ 0 & u_{jj} & \mu_j^T \\ 0 & 0 & U_{33} \end{pmatrix}.$$

Equating the coefficients in the j^{th} column, we get

$$a_j = U_{11}^T u_j \text{ and}$$

$$a_{jj} = u_j^T u_j + u_{jj}^2.$$

Assuming that we have already computed U_{11}^T , then we compute both u_j and u_{jj} directly from the equations

$$U_{11}^T u_j = a_j \text{ and}$$

$$u_{jj}^2 = a_{jj} - u_j^T u_j.$$

LINPACK routine SPOFA implements operation using

```
DO 30 J = 1, N
  INFO = J
  S = 0.0E0
  JM1 = J - 1
  IF (JM1 .LT. 1) GO TO 20
  DO 10 K = 1, JM1
    T = A(K,J) - SDOT(K-1,A(1,K),1,A(1,J),1)
    T = T/A(K,K)
    A(K,J) = T
    S = S + T*T
  10  CONTINUE
  20  CONTINUE
  S = A(J,J) - S
C   .....EXIT
  IF (S .LE. 0.0E0) GO TO 40
  A(J,J) = SQRT(S)
  30  CONTINUE
```

The operation can be implemented using Level 2 BLAS instead.

```
DO 10 J = 1, N
  CALL STRSV( 'Upper', 'Transpose', 'Non-unit', J-1, A, LDA,
$           A(1,J), 1 )
  S = A(J,J) - SDOT( J-1, A(1,J), 1, A(1,J), 1 )
  IF( S.LE.ZERO ) GO TO 20
  A(J,J) = SQRT( S )
10 CONTINUE
```

The call to STRSV (which solves a triangular system of equations) has replaced the loop over K which made several calls to the Level 1 BLAS routine SDOT.

This change means much higher performance on a vector computer, a breed that has nearly disappeared. It makes almost no difference on RISC and x86 based CPUs, however. ☹

Block forms of Cholesky factorization work well on both types of computers. We rewrite the equations again in the form

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ \cdot & A_{22} & A_{23} \\ \cdot & \cdot & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ U_{12}^T & U_{22}^T & 0 \\ U_{13}^T & U_{23}^T & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix}.$$

We equate submatrices in the second block of columns to get

$$A_{12} = U_{11}^T U_{12} \text{ and}$$

$$A_{22} = U_{12}^T U_{12} + U_{22}^T U_{22}.$$

Assuming that we have already computed U_{11}^T , then we compute U_{12} as the solution to $U_{11}^T U_{12} = A_{12}$ using a call to the Level 3 BLAS function STRSM.

We compute U_{22} from $U_{22}^T U_{22} = A_{22} - U_{12}^T U_{12}$ by updating the symmetric submatrix A_{22} and then doing its Cholesky factorization. Due to a Fortran defect (no recursion!), this requires calling another function to do the work (SPOTF2).

The code can be written in the following form:

```
DO 10 J = 1, N, NB
  JB = MIN( NB, N-J+1 )
  CALL STRSM( 'Left', 'Upper', 'Transpose', 'Non-unit', J-1, JB,
$           ONE, A, LDA, A( 1, J ), LDA )
  CALL SSYRK( 'Upper', 'Transpose', JB, J-1, -ONE, A( 1, J ), LDA,
$           ONE, A( J, J ), LDA )
  CALL SPOTF2( 'Upper', JB, A( J, J ), LDA, INFO )
  IF( INFO.NE.0 ) GO TO 20
10 CONTINUE
```

This is fast, but not fast enough on all computers. It turns out that the implementation in LAPACK uses Level 3 BLAS matrix-matrix multiplication to get the best performance. So the code actually is of the form,

```

DO 10 J = 1, N, NB
  JB = MIN( NB, N-J+1 )
  CALL SSYRK( 'Upper', 'Transpose', JB, J-1, -ONE,
$          A( 1, J ), LDA, ONE, A( J, J ), LDA )
  CALL SPOTF2( 'Upper', JB, A( J, J ), LDA, INFO )
  IF( INFO.NE.0 ) GO TO 30
  IF( J+JB.LE.N ) THEN
    CALL SGEMM( 'Transpose', 'No transpose', JB, N-J-JB+1,
$          J-1, -ONE, A( 1, J ), LDA, A( 1, J+JB ),
$          LDA, ONE, A( J, J+JB ), LDA )
    CALL STRSM( 'Left', 'Upper', 'Transpose', 'Non-unit',
$          JB, N-J-JB+1, ONE, A( J, J ), LDA,
$          A( J, J+JB ), LDA )
    END IF
10  CONTINUE

```

What is really going on?

- In many linear algebra algorithms involving matrices, there are so-called i , j , and k variants depending on the order of loops involving the three indices. There are 6 interesting variants based on memory access (see next two slides).
- Linpack used the j variant whereas the i variant turns out to be the winner for today's CPUs. The j variant requires solving a triangular system of equations. The i variant requires a matrix-matrix multiplication. Both require $O(n^2)$ operations with similar constants, but the matrix-matrix multiply is one of the targets for high performance by recent CPU architects whereas triangular solves is not a target at all.
- 30-40 years of assumptions about loop orderings were invalidated thanks to a (simple) design change in CPUs.

Six variants of LU triple loops:

1. ijk Loop: A - by column (ddot)	2. ikj Loop: A - by row (daxpy)
<pre> for $i = 2, n$ for $j = 2, i$ $\ell_{i,j-1} = a_{i,j-1}/a_{j-1,j-1}$ for $k = 1, j - 1$ $a_{ij} = a_{ij} - \ell_{ik}a_{kj}$ endfor endfor for $j = i + 1, n$ for $k = 1, i - 1$ $a_{ij} = a_{ij} - \ell_{ik}a_{kj}$ endfor endfor endfor </pre>	<pre> for $i = 2, n$ for $k = 1, i - 1$ $\ell_{ik} = a_{ik}/a_{kk}$ for $j = k + 1, n$ $a_{ij} = a_{ij} - \ell_{ik}a_{kj}$ endfor endfor endfor </pre>

<p>3. <i>jik</i> Loop: A - by column (ddot)</p> <pre> for j = 2, n for p = j, n $\ell_{p,j-1} = a_{p,j-1}/a_{j-1,j-1}$ endfor for i = 2, j for k = 1, i - 1 $a_{ij} = a_{ij} - \ell_{ik}a_{kj}$ endfor endfor for i = j + 1, n for k = 1, j - 1 $a_{ij} = a_{ij} - \ell_{ik}a_{kj}$ endfor endfor endfor </pre>	<p>4. <i>jki</i> Loop: A - by column (daxpy)</p> <pre> for j = 2, n for p = j, n $\ell_{p,j-1} = a_{p,j-1}/a_{j-1,j-1}$ endfor for k = 1, j - 1 for i = k + 1, n $a_{ij} = a_{ij} - \ell_{ik}a_{kj}$ endfor endfor endfor </pre>
<p>5. <i>kij</i> Loop: A - by row (daxpy)</p> <pre> for k = 1, n - 1 for i = k + 1, n $\ell_{ik} = a_{ik}/a_{kk}$ for j = k + 1, n $a_{ij} = a_{ij} - \ell_{ik}a_{kj}$ endfor endfor endfor </pre>	<p>6. <i>kji</i> Loop: A - by column (daxpy)</p> <pre> for k = 1, n - 1 for p = k + 1, n $\ell_{pk} = a_{pk}/a_{kk}$ endfor for j = k + 1, n for i = k + 1, n $a_{ij} = a_{ij} - \ell_{ik}a_{kj}$ endfor endfor endfor </pre>

OpenACC <http://www.openacc-standard.org>

A directive based system for C, C++, and Fortran to expose parallelism in codes so that compilers can do the hard work of mapping the code to an accelerator, such as a GPU.

This is part of an OpenMP working group. The 1.0 standard was approved in November, 2011 shortly before SC'11. The group includes CAPS, Cray, NVIDIA, and PGI. The first compiler with OpenACC should be available during 1H2012. It is largely based on the PGI directives.

OpenACC can be built on proprietary accelerator software, CUDA, or OpenCL. Innovative additional functionality seems to be encouraged by the working group.

Goals of OpenACC include

- It enables an easy path for developers to start reaping the benefits from powerful many-core accelerators like GPUs.
- It permits a common code base for accelerated and non-accelerator enabled systems. OpenACC directives can be ignored on non-accelerator enabled systems.
- The specification allows for potential implementations on different types and brands of accelerators (CPUs, GPUs)
- It provides an incremental path for moving legacy applications to accelerators that may disturb the existing code less than other approaches.
- It allows programmer tools to focus on supporting a common accelerator standard with their own unique extensions.
- It provides a quick way onto accelerators for scientists who do not want to be trapped by a hardware vendor.

Execution model

Host/accelerator regions of code. Most code runs on the host while compute intensive parts run on the accelerator. The overall code will still work if no accelerator is present.

Two-three levels of parallelism may be supported (depends on the accelerator):

- Coarse grained – all is executed on all parallel units.
- Fine grained – multiple threads execute per execution unit.
- SIMD – vector mode

Synchronization may exist in each of these modes.

In loops, data dependencies must be reprogrammed to allow the parallelism to work correctly.

Memory model

Memory on the accelerator is independent of the host. Memory copying is used to get data to the write device's memory, which is slow. Re-use of data on the accelerator is imperative.

In the OpenACC model, data movement between the memories is implicit and managed by the compiler based on directives. The programmer must be aware of the potentially separate memories for many reasons, including but not limited to:

- Memory bandwidth between host memory and device memory determines the level of compute intensity required to effectively accelerate a given region of code.
- The limited device memory size may prohibit offloading of regions of code that operate on very large amounts of data.

Many GPUs today have little or no memory coherence. Programmers have to guarantee coherence through the code written.

Memory caches are also interesting, with some using hardware, some using software, and some using read only caches for constants. In OpenCL and CUDA the programmer has to manage everything correctly based on the properties of the actual accelerator being used. This can vary quite a lot, unfortunately.

In OpenACC, directives are used to manage cache memory, which is much easier for programmers, but may not actually do what you think it will on all accelerators (based on the features of the device).

Data consists of scalars, arrays, and subarrays. Using *lower:upper* constructs in arrays, subarrays can be defined and used in directives. The compiler will fill in missing *lower* and *upper* values if they are known at compile time. The subarray feature is similar to Fortran's subarray syntax. This is a powerful way to move only what is needed to an accelerator instead of entire arrays.

OpenACC Directives

C/C++:

`#pragma acc directive-name[clause[[,]clause]...]new-line`

Fortran (but not in PURE or ELEMENTAL procedures):

`!$acc directive-name[clause[[,]clause]...]`

Macro `_OPENACC` is defined as a date of the form `yyyymm`, where `yyyy` is the year and `mm` is the month of the specification. This can be used for condition compilation.

Before execution of any OpenACC routine or directive is executed, certain internal system variables are set, either by the runtime system or the user. Once an OpenACC routine or directive is executed, these variables cannot be changed. Of importance are the environment variables

- ACC_DEVICE_TYPE
- ACC_DEVICE_NUM

They can be set using `acc_set_device_type()` and `acc_set_device_num()`, respectively. The values can be retrieved using `acc_get_device_type()` and `acc_get_device_num()`, respectively.

Parallel Directive

`#pragma acc parallel [clause[,]clause...]new-line
structured block`

where there are many, many clauses, some similar to ones in OpenMP, but others specific to accelerators.

`if(condition)`

- The compiler will generate two copies of the construct, one copy to execute on the accelerator and one copy to execute on the host. When the condition in the if clause evaluates to zero in C or C++, or .false. in Fortran, the host copy will be executed. When the condition evaluates to nonzero in C or C++, or .true. in Fortran, the accelerator copy will be executed.

`async` [(scalar-integer-expression)]

- The parallel or kernels region will be executed by the accelerator device asynchronously while the host process continues with the code following the region.
- The argument must be result in an integer and is used in `wait` directives.

`num_gangs`(scalar-integer-expression)

- The value of the integer expression defines the number of parallel gangs that will execute the region.
- If the clause is not specified, an implementation-defined default will be used.

`num_workers(scalar-integer-expression)`

- The value of the integer expression defines the number of workers within each gang that will execute the region.
- If the clause is not specified, an implementation-defined default will be used (*probably* 1).

`vector_length(scalar-integer-expression)`

- The value of the integer expression defines the vector length to use for vector or SIMD operations within each worker of the gang.
- If the clause is not specified, an implementation-defined default will be used.
- This vector length will be used for loops annotated with the vector clause on a loop directive, and for loop automatically vectorized by the compiler.

- There may be implementation defined limits on the allowed values for the vector length expression.

reduction(operator:list)

- This is used to do a reduction operation, e.g., an inner product with a scalar result from two shared global vectors so that operation would be + in this case.
- The actual operators allowed in C/C++ are +, *, max, min, &, |, ^, &&, and ||.

copy(list)

- Used to declare that the variables, arrays, or subarrays in the list have values in the host memory that need to be copied to the device memory, and are assigned values on the accelerator that need to be copied back to the host.

copyin(list)

- Used to declare that the variables, arrays, or subarrays in the list have values in the host memory that need to be copied to the device memory.

copyout(list)

- Used to declare that the variables, arrays, or subarrays in the list are assigned or contain values in the device memory that need to be copied back to the host memory at the end of the accelerator region.

`create(list)`

- Used to declare that the variables, arrays, or subarrays in the list need to be allocated (created) in the device memory, but the values in the host memory are not needed on the accelerator, and any values computed and assigned on the accelerator are not needed on the host.
- No data in this clause will be copied between the host and device memories.

`present(list)`

- Used to tell the implementation that the variables or arrays in the list are already present in the accelerator memory due to data regions that contain this region, perhaps from procedures that call the procedure containing this construct.

- The implementation will find and use that existing accelerator data.
- If there is no containing data region that has placed any of the variables or arrays on the accelerator, the program will halt with an error.

`present_or_copy(list)`

- Also called `pcopy`.
- Works like either a `present` or `copy` (copies both ways).

`present_or_copyin(list)`

- Also called `pcopyin`.
- Works like either a `present` or `copyin`.

`present_or_copyout(list)`

- Also called `pcopyout`.
- Works like either a `present` or `copyout`.

`present_or_create(list)`

- Also called `create`.
- Works like either a `creat` or `copy`.

`deviceptr(list)`

- Declare that the pointers in the list are device pointers, so the data need not be allocated or moved between the host and device for this pointer.

`private(list)`

- Each thread gets its own copy, which is uninitialized.
- Value lost at end of block.

`firstprivate(list)`

- Listed variables have their own copy and are initialized with the value from before the block.

A gang of workers is created on the accelerator. The number of workers per gang is constant throughout execution. A worker inside each gang is executed in a not predictable order, so the code must handle this correctly. Simple algorithms that are not data order dependent are easiest to implement for this model.

Notes:

- If `async` is not present, then a barrier is implied at the end of the region.

- An array or variable of aggregate data type referenced in the parallel construct that does not appear in a data clause for the construct or any enclosing data construct will be treated as if it appeared in a **present_or_copy** clause for the parallel construct.
- A scalar variable referenced in the parallel construct that does not appear in a data clause for the construct or any enclosing data construct will be treated as if it appeared in a **private** clause (if not live-in or live-out) or a **copy** clause for the parallel construct.

Restrictions:

- OpenACC parallel regions may not contain other parallel regions or kernels regions.
- A program may not branch into or out of an OpenACC parallel construct.

- A program must not depend on the order of evaluation of the clauses, or on any side effects of the evaluations.
- At most one if clause may appear. In Fortran, the condition must evaluate to a scalar logical value; in C or C++, the condition must evaluate to a scalar integer value.

Kernels Directive

```
#pragma acc kernels [clause[[,]clause]...] new-line  
structured block
```

This creates a set of kernels on the accelerator, typically each one being a loop nest. The kernels are executed on the accelerator in order of appearance in the block. The number and configuration of

gangs of workers and vector length may be different for each kernel. The list of clauses includes

- `if(condition)`
- `async [(scalar-integer-expression)]`
- `copy(list)`
- `copyin(list)`
- `copyout(list)`
- `create(list)`
- `present(list)`
- `present_or_copy(list)`
- `present_or_copyin(list)`
- `present_or_copyout(list)`
- `present_or_create(list)`
- `deviceptr(list)`

Restrictions:

- OpenACC kernels regions may not contain other parallel regions or kernels regions.
- A program may not branch into or out of an OpenACC kernels construct.
- A program must not depend on the order of evaluation of the clauses, or on any side effects of the evaluations.
- At most one if clause may appear. In Fortran, the condition must evaluate to a scalar logical value. In C or C++, the condition must evaluate to a scalar integer value.

Data Directive

`#pragma acc data [clause[[,]clause]...] new-line
structured block`

The data construct defines scalars, arrays and subarrays to be allocated in the device memory for the duration of the region, whether data should be copied from the host to the device memory upon region entry, and copied from the device to host memory upon region exit.

The same clauses are used as in the kernels directive.

Restrictions:

- None

Host_Data Directive

```
#pragma acc host_data [clause[[,]clause]...] new-line  
structured block
```

The host_data construct makes the address of device data available on the host. The only clause is

`use_device(list)`

- Use the device address of any variable or array in the list in code within the construct.
- This may be used to pass the device address of variables or arrays to optimized procedures written in a lower level API.
- The variables or arrays in list must be present in the accelerator memory due to data regions that contain this construct.

Loop Directive

```
#pragma acc loop [clause[,]clause...] new-line  
for loop
```

The OpenACC loop directive applies to a loop which must immediately follow this directive. The loop directive can describe what type of parallelism to use to execute the loop and declare loop-private variables and arrays and reduction operations. Some clauses are only valid in the context of a parallel region, and some only in the context of a kernels region (see the descriptions below):

`collapse(n)`

- Used to specify how many tightly nested loops are associated with the loop construct. The argument to the collapse clause must be a constant positive integer expression. If no collapse clause is present, only the immediately following loop is associated with the loop directive.
- If more than one loop is associated with the loop construct, the iterations of all the associated loops are all scheduled according to the rest of the clauses. The trip count for all loops associated with the collapse clause must be computable and invariant in all the loops.
- It is implementation-defined whether a gang, worker or vector clause on the directive is applied to each loop, or to the linearized iteration space.

gang [(scalar-integer-expression)]

- In an accelerator parallel region, the gang clause specifies that the iterations of the associated loop or loops are to be executed in parallel by distributing the iterations among the gangs created by the parallel construct. *No argument is allowed.* The loop iterations must be data independent, except for variables specified in a reduction clause.
- In an accelerator kernels region, the gang clause specifies that the iterations of the associated loop or loops are to be executed in parallel across the gangs created for any kernel contained within the loop or loops. If an argument is specified, it specifies how many gangs to use to execute the iterations of this loop.

worker [(scalar-integer-expression)]

- In an accelerator parallel region, the worker clause specifies that the iterations of the associated loop or loops are to be executed in parallel by distributing the iterations among the multiple workers within a single gang. *No argument is allowed.* The loop iterations must be data independent, except for variables specified in a reduction clause. It is implementation-defined whether a loop with the worker clause may contain a loop containing the gang clause.
- In an accelerator kernels region, the worker clause specifies that the iterations of the associated loop or loops are to be executed in parallel across the workers within the gangs created for any kernel contained within the loop or loops. If an argument is specified, it specifies how many workers to use to execute the iterations of this loop.

vector [(scalar-integer-expression)]

- As before.

seq

- Specifies that the associated loop or loops are to be executed sequentially by the accelerator; this is the default in an accelerator parallel region. This clause will override any automatic compiler parallelization or vectorization.

independent

- In an accelerator kernels region, this tells the compiler that the iterations of this loop are data-independent with respect to each other. This allows the compiler to generate code to execute the iterations in parallel with no synchronization.

- It is a programming error to use the independent clause if any iteration writes to a variable or array element that any other iteration also writes or reads, except for variables in a reduction clause.

`private(list)`

- As before.

`reduction(operator:list)`

- As before.

In a parallel region, a loop directive with no `gang`, `worker`, or `vector` clauses allows the implementation to automatically select whether to execute the loop across gangs, workers within a gang, or whether to execute as vector operations. The implementation

may also choose to use vector operations to execute any loop with no loop directive, using classical automatic vectorization.

Cache Directive

`#pragma acc cache(list) new-line`

The cache directive may appear at the top of (inside of) a loop. It specifies array elements or subarrays that should be fetched into the highest level of the cache for the body of the loop.

The entries in **list** must be single array elements or simple subarray.

Declare Directive

`#pragma acc declare declclause [[,] declclause]... new-line`

This directive is used in the declaration section of a Fortran subroutine, function, or module, or following a variable declaration in C or C++. It can specify that a variable or array is to be allocated in the device memory for the duration of the implicit data region of a function, subroutine or program, and specify whether the data values are to be transferred from the host to the device memory upon entry to the implicit data region, and from the device to the host memory upon exit from the implicit data region. These directives create a visible device copy of the variable or array.

The declclause arguments can be

`copy(list)`

`copyin(list)`

`copyout(list)`

`create(list)`

`present(list)`

`present_or_copy(list)`

`present_or_copyin(list)`

`present_or_copyout(list)`

`present_or_create(list)`

`deviceptr(list)`

`device_resident(list)`

- Specifies that the memory for the named variables should be allocated in the accelerator device memory, not in the host memory.

Restrictions:

- A variable or array may appear at most once in all the clauses of declare directives for a function, subroutine, program, or module.
- Subarrays are not allowed in declare directives.
- If a variable or array appears in a declare directive, the same variable or array may not appear in a data clause for any construct where the declaration of the variable is visible.
- The compiler may pad dimensions of arrays on the accelerator to improve memory alignment and program performance.

- In Fortran:
 - Assumed-size dummy arrays may not appear in a declare directive.
 - Pointer arrays may be specified, but pointer association is not preserved in the device memory.

Update Directive

`#pragma acc update clause [[,] clause]... new-line`

with clauses

`host(list)`

`device(list)`

`if(condition)`

`async` [(scalar-integer-expression)]

The list argument to an update clause is a comma-separated collection of variable names, array names, or subarray specifications. Multiple subarrays of the same array may appear in a list. The effect of an update clause is to copy data from the accelerator device memory to the host memory for update host, and from host memory to accelerator device memory for update device. The updates are done in the order in which they appear on the directive. There must be a visible device copy of the variables or arrays that appear in the `host` or `device` clauses. At least one `host` or `device` clause must appear.

Restrictions:

- The update directive is executable. It must not appear in place of the statement following an if, while, do, switch, or label in C or C++, or in place of the statement following a logical if in Fortran.
- A variable or array which appears in the list of an update directive must have a visible device copy.

Wait Directive

#pragma acc wait [(scalar-integer-expression)] new-line

The argument, if specified, must be an integer expression. The host thread will wait until all asynchronous activities that had an async clause with an argument with the same value have completed.

If no argument is specified, the host process will wait until all asynchronous activities have completed.

If there are two or more host threads executing and sharing the same accelerator device, a wait directive will cause the host thread to wait until at least all of the appropriate asynchronous activities initiated by that host thread have completed. There is no guarantee that all the similar asynchronous activities initiated by some other host thread will have completed.

Runtime Library Definitions

All definitions are in header or module files. For C/C++:

`openacc.h`

All functions are `extern` with “C” linkage.

For Fortran:

`openacc_lib.h` and module `openacc`

that contains interfaces to functions, as well as integer parameters for argument types, `openacc_version` (same as `_OPENACC`), and device types.

The device data type is

`acc_device_t` (C/C++)
`integer(kind=acc_device_kind)` (Fortran).

Many functions return a value indicating a device type, which are

- `acc_device_none`
- `acc_device_default`
- `acc_device_host`
- `acc_device_not_host`

`acc_device_default` is never returned by current OpenACC functions, but is used to tell a function in the runtime library to use the default device type.

Runtime Library Routines

Only the C/C++ interfaces to the runtime library are described here. The Fortran interfaces are described at the OpenACC web site.

`acc_get_num_devices`

`void acc_set_device_type (acc_device_t);`

The `acc_get_num_devices` routine returns the number of accelerator devices of the given type attached to the host. The argument tells what kind of device to count.

`acc_set_device_type`

`void acc_set_device_type (acc_device_t);`

Sets which type of device to use among those available.

Restrictions:

- Should be called before any interaction with accelerators or after a call to `acc_shutdown`.
- If the device is not available, good luck on what happens next.
- Do not call with a different device type before calling `acc_shutdown` beforehand. Good luck otherwise.
- If a region requires a different type of accelerator than what you use as a device in this call, good luck.

`acc_get_device_type`

`acc_device_t acc_get_device_type (void);`

Returns a value corresponding to the device type that will be used in the next region or kernel. This is only useful if you have a choice of accelerators for the next region or kernel.

Restrictions:

- This routine may not be called during execution of an accelerator parallel or kernels region.
- If the device type has not yet been selected, the value `acc_device_none` will be returned.

`acc_set_device_num`

`void acc_set_device_num(int, acc_device_t);`

Sets which device to use among those attached of the given type. If the value of devicenum is zero, the runtime will revert to its default behavior. If the value of the second argument is zero, the selected device number will be used for all attached accelerator types.

Restrictions:

- Do not call during execution of an accelerator parallel, kernels, or data region.
- If the value of devicenum is greater than the value returned by `acc_get_num_devices` for that device type, good luck.

- Calling `acc_set_device_num` implies a call to `acc_set_device_type` with that device type argument.

`acc_get_device_num`

```
int acc_get_device_num( acc_device_t );
```

This returns an integer corresponding to the device number of the specified device type that will be used to execute the next accelerator parallel or kernels region.

Restriction:

- Do not call during execution of an accelerator parallel or kernels region.

`acc_async_test`

`int acc_async_test(int);`

The argument must be an integer expression. If that value appeared in one or more async clauses, and all such asynchronous activities have completed, then a nonzero value is returned. Otherwise a zero is returned.

Restriction:

- Do not call during execution of an accelerator parallel or kernels region.

acc_async_test_all

```
int acc_async_test_all( );
```

If all outstanding asynchronous activities have completed, then a nonzero value is returned. Otherwise a zero is returned.

Restriction:

- Do not call during execution of an accelerator parallel or kernels region.

`acc_async_wait`

`void acc_async_wait(int);`

If argument value appeared in one or more async clauses, the routine will not return until the latest such asynchronous activity has completed. If there are two or more host threads sharing the same accelerator, the routine will return only if all matching asynchronous activities initiated by this host thread have completed. There is no guarantee that all matching asynchronous activities initiated by other host threads have completed.

Restriction:

- Do not call during execution of an accelerator parallel or kernels region.

`acc_async_wait_all`

`void acc_async_wait_all();`

The routine will not return until the all asynchronous activities have completed. If there are two or more host threads sharing the same accelerator, the routine will return only if all matching asynchronous activities initiated by this host thread have completed. There is no guarantee that all matching asynchronous activities initiated by other host threads have completed.

Restriction:

- Do not call during execution of an accelerator parallel or kernels region.

`acc_init`

`void acc_init (acc_device_t);`

Initializes the runtime for a device type and also calls `acc_set_device`. This function can be used to isolate any initialization cost from the computational cost when timing.

Restrictions:

- Do not call in an accelerator parallel or kernels region.
- If the device type specified is not available, good luck.
- If the routine is called more than once without an intervening `acc_shutdown` call, good luck.

- If some accelerator regions are compiled to only use one device type and you call this routine with a different device type, good luck.

`acc_shutdown`

```
void acc_shutdown ( acc_device_t );
```

The program is disconnected from the accelerator device.

Restriction:

- Do not call during execution of an accelerator region.

`acc_on_device`

```
int acc_on_device ( acc_device_t );
```

This routine may be used to execute different paths depending on whether the code is running on the host or on some accelerator. If the `acc_on_device` routine has a compile-time constant argument, it evaluates at compile time to a constant. The argument must be one of the defined accelerator types.

If the argument is `acc_device_host`, then outside of an accelerator parallel or kernels region, or in an accelerator parallel or kernels region that is executed on the host processor, this routine will evaluate to nonzero. Otherwise it returns 0. Fortran uses `.true.` and `.false.` instead.

acc_malloc

```
void* acc_malloc ( size_t );
```

This allocates memory on the accelerator device. Pointers assigned from this function may be used in `deviceptr` clauses to tell the compiler that the pointer target is resident on the accelerator.

acc_free

```
void acc_free ( void* );
```

This frees previously allocated memory on the accelerator device; the argument should be a pointer value that was returned by a call to `acc_malloc`.

Computer Accelerator Observations

Every decade since the 1950's a small collection of computer vendors have had one or two ideas that lead to 1-2 orders of magnitude speedup.

As a result,

- New mathematical algorithms usually are created that provide a speedup equal to the hardware acceleration.
- New mathematical analysis usually occurs.
- Combination is a good symbiotic relationship.

The latest, greatest hardware accelerator is the General Purpose - Graphics Processing Unit (GP-GPU or just GPU).

History demonstrates that technology that does not migrate into the CPU disappears over time or morphs into something that does.

A few historical highlights

- Late 1950's: First supercomputer proposed, IBM 7030 (failed, but led to many successes)
- Early 60's: Multibank memories (roaring success)
- Late 60's/early 70's: Illiac IV → Thinking Machines CM1 (gone)
- Mid 70's: array processors → vector computers, (disappearing for years, slowly).
- Early 80's: array processors → VLIW (mostly gone)
- Mid 80's: Intel iPSC → called blade servers now (roaring success)
- Mid 80's: Thinking Machines CM-1 → GP-GPUs? (reborn?)

- Late 80's: RISC → out of order execution (RISC pointing to nite-nite, sucesor alive and well)
- Late 80's: SMPs → CMPs (alive and very well)
- Mid to late 90's: Multilevel memory caching (alive and very well, unfortunately)
 - NUMA
 - Multi-core
 - Many-core
 - No speed up (slow down!) after some number of cores
- Mid 00's: FPGA IEEE double precision floating point works
 - Major leap forward in technology
 - Forced GPU makers to follow suit within a year
 - New software is created to make programming these systems far easier than a few years ago.

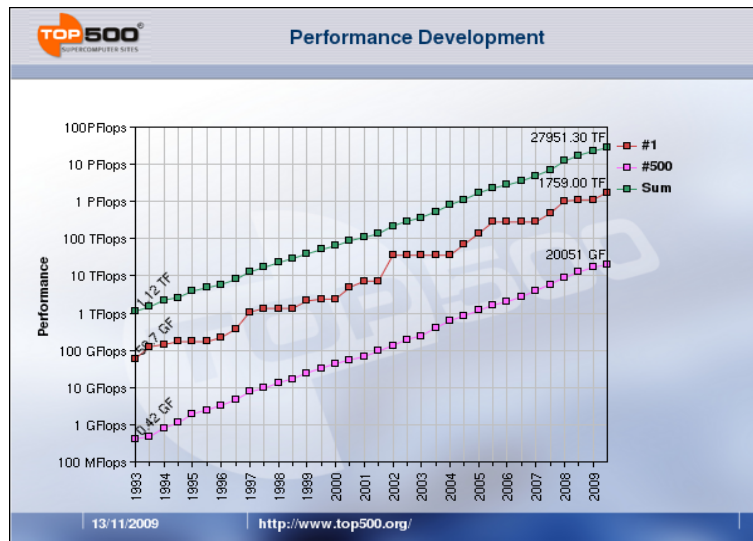
- Lots of companies in business at SC'xx are out of business at SC'xx+1 or SC'xx+2 (just like the good old days of parallel computing companies).
- Who is missing? CPU designers signed off on traditional supercomputers a very long time ago.
 - The Willy Sutton philosophy from the 1920-1950's: When asked why he robbed banks for a living, he replied, "Because that's where the money is."
- So where is the money today in computers?
 - Relational databases
 - Web services
 - Transaction services
 - This is where \$o(1,000,000,000,000) of business a year is versus \$O(10,000,000,000)
- GP-GPU designers operate in a *Wild West* mode.

Can a GP-GPU replace a supercomputer?

Maybe, but they can surely augment traditional supercomputers.

Consider NVIDIA Tesla C2050/C2070 cards:

- Double Precision floating point : **0.515 Tflops**
- Single Precision floating point: **1.030 Tflops**
- *Enables over 4 Tflops on a workstation*



Advantages:

- Powerful computing performance (1.03 Tflops) with a low price
- Light weight threading
- Shared memory
- Programmable in C, C++, or Fortran (+ weird programming)

Disadvantages:

- Bottleneck on memory bandwidth (PCI Express 2.0 x16: 8 GB/sec)
- Block communication is allowed only through global memory (Memory bandwidth: 102 GB/sec)
- Has to be hand optimized by a programmer until recently when pragmas were developed by PGI and Intel, followed by OpenACC standard within OpenMP framework.

In comparison to recent quad core CPUs, GPUs claim to be about 1/10-th the cost and 1/20-th the power per Teraflop.

CPUs

- 2-4 cores prevalent, 8-16 available, but not as common
- Good use of caches, fast memory bandwidth, and large memories

GPUs

- 240-448 cores with 1-4 GPUs on a card
- Double precision slow (2-11X slower than single precision)
- Small random access memory or data streaming exist
- Memory bandwidth a serious problem

Properties of GP-GPUs

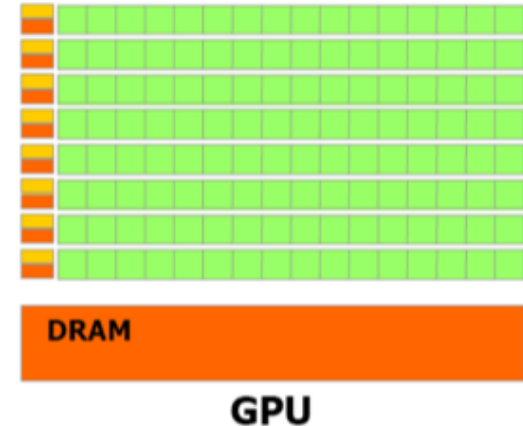
Re-use of data with minimal data transfers between CPU memory and GPU memory *critical*

- Data flow/stream style, ala ATI GPUs or Cell Processor
- Multi-bank, random access, ala NVIDIA GPUs
 - Pull data from 16 memory banks every cycle
 - Linear slow down with less than 16 memory bank access
 - Sounds like a 1960's computing hardware accelerator
- **Good candidates for GP-GPUs**
 - Time dependent PDEs and nonlinear problems
 - Iterative methods with a *lot* of iterations
 - *Anything that vectorized well in the Cray 2 era!*

GP-GPU versus CPU

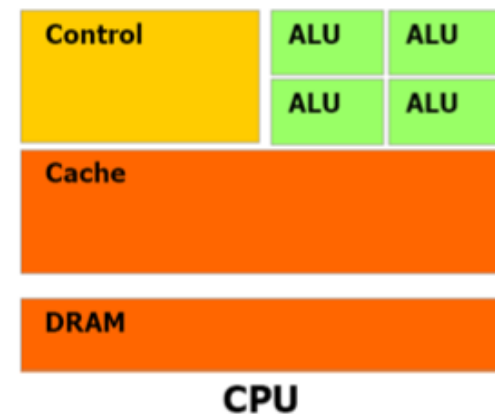
GP-GPU

- Devotes more transistors to data processing than a CPU
- Simple scalar cores with massive multithreading to hide latency
- No coherent cache, just shared memory

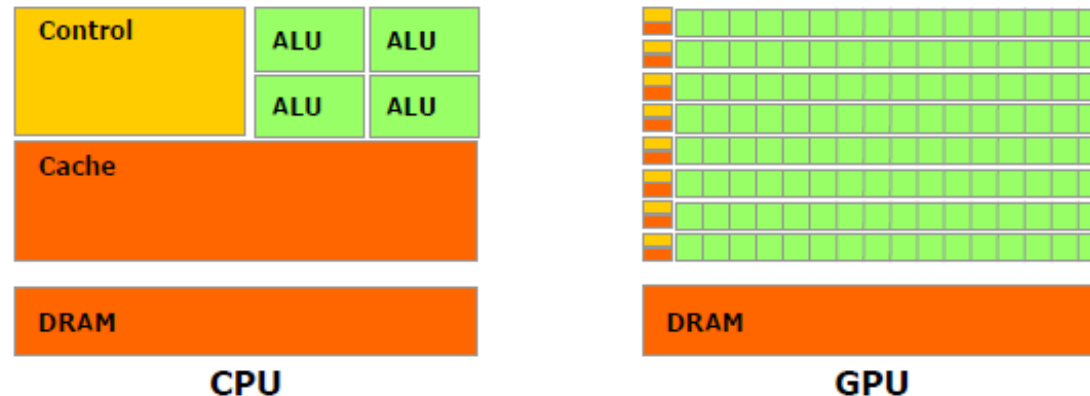


CPU

- Complicated deeply pipelined out of order execution cores
- Big L1, L2, ... caches to hide latency to main memory



What is a GP-GPU?



The reason behind the discrepancy in floating-point capability between a CPU and a GP-GPU is that the GP-GPU is specialized for compute-intensive, highly parallel computation – exactly what graphics rendering is about – and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control.

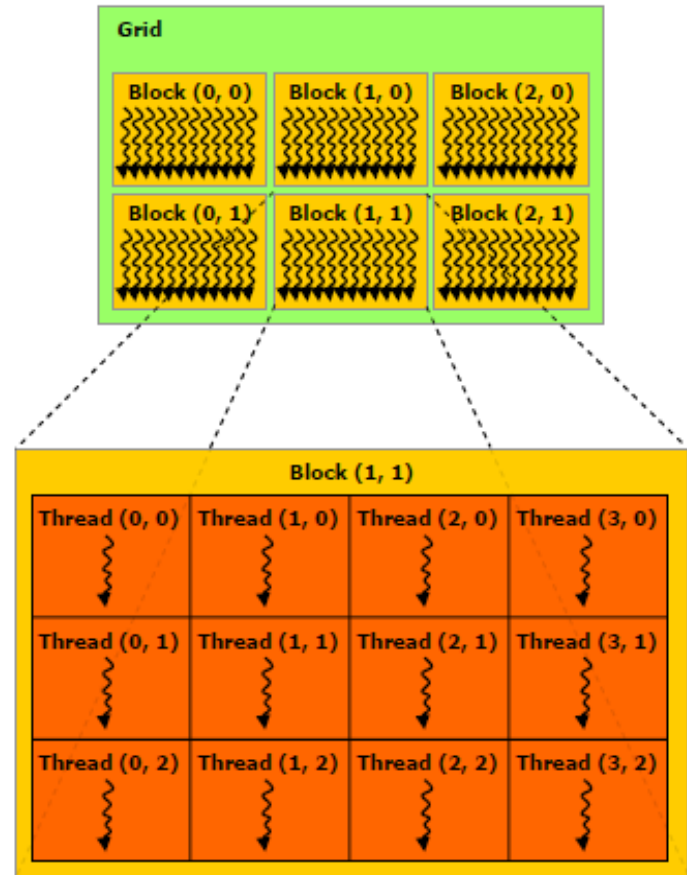
What is a GP-GPU? (NVIDIA specific)

Blocks

- Blocks are organized into a one-, two- or three-dimensional grid. (up to $512 \times 512 \times 64$)
- The number of the blocks are usually dictated by the size of the data or the number of processors.
- Each block contains up to 512 threads.

Threads

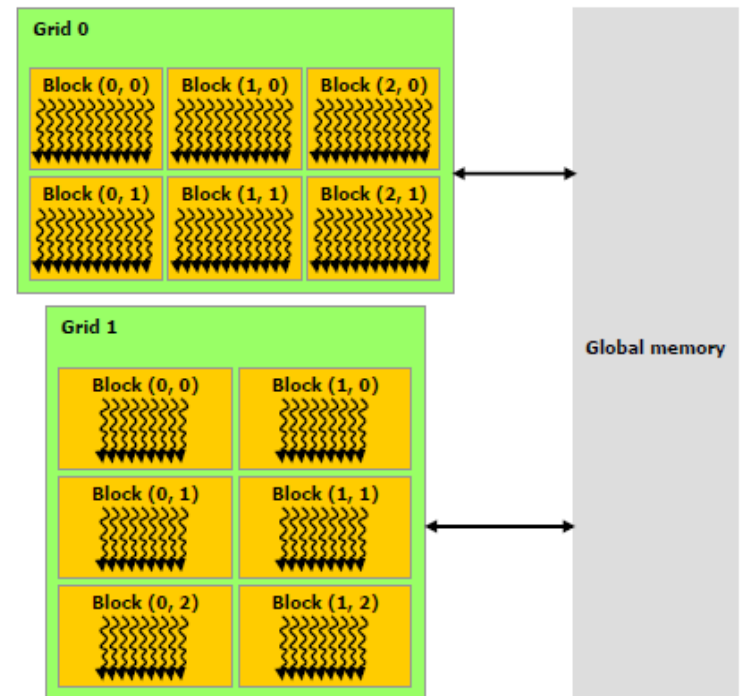
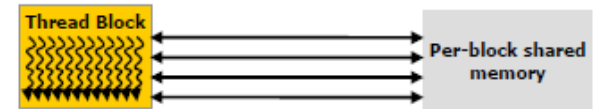
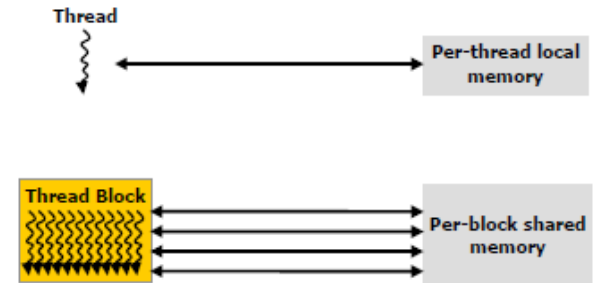
- Threads reside on the same processor core.
- Threads in a same block share the shared memory



Memory hierarchy

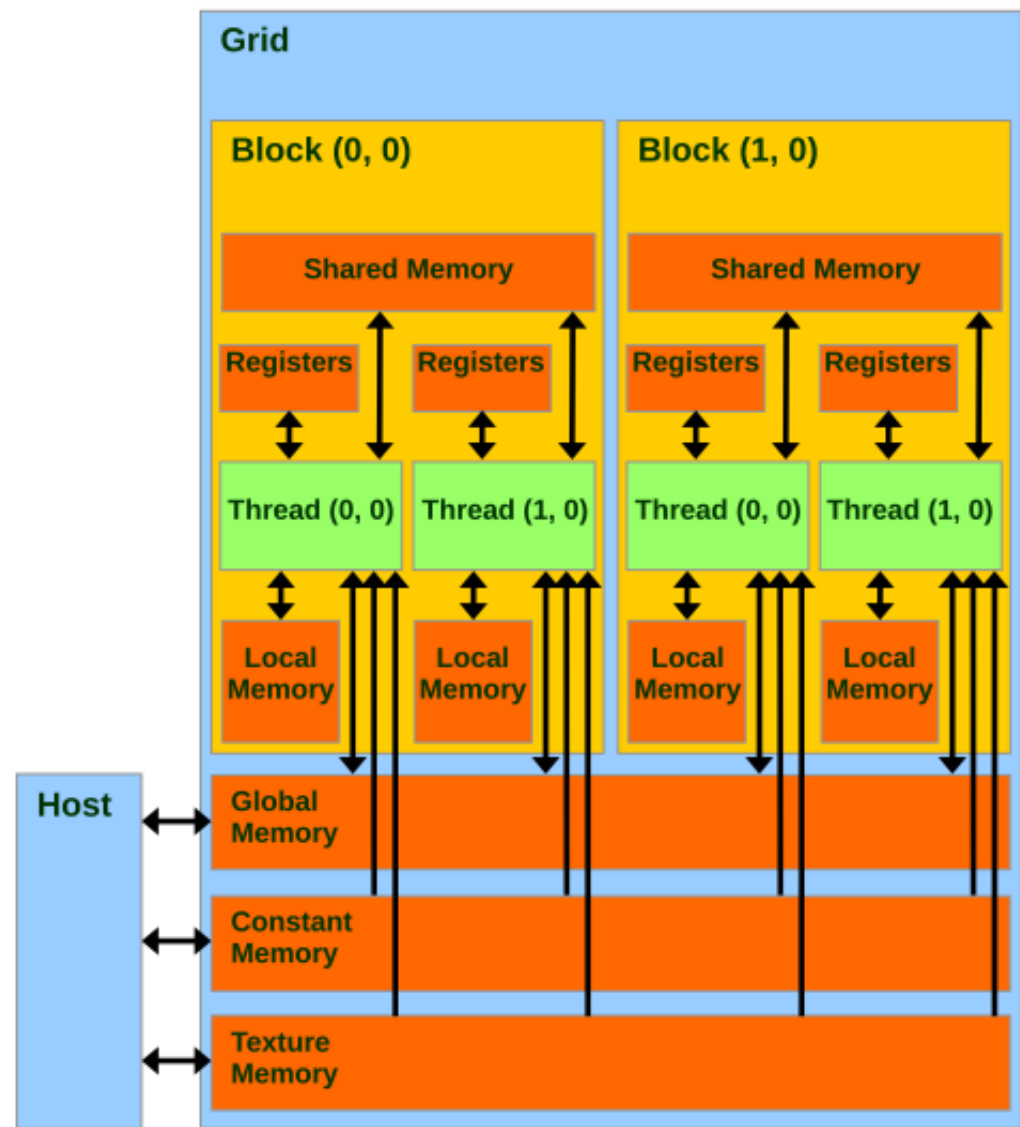
- Local memory
- Shared memory
- Global memory
- Constant memory
- Texture memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	No	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	No	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

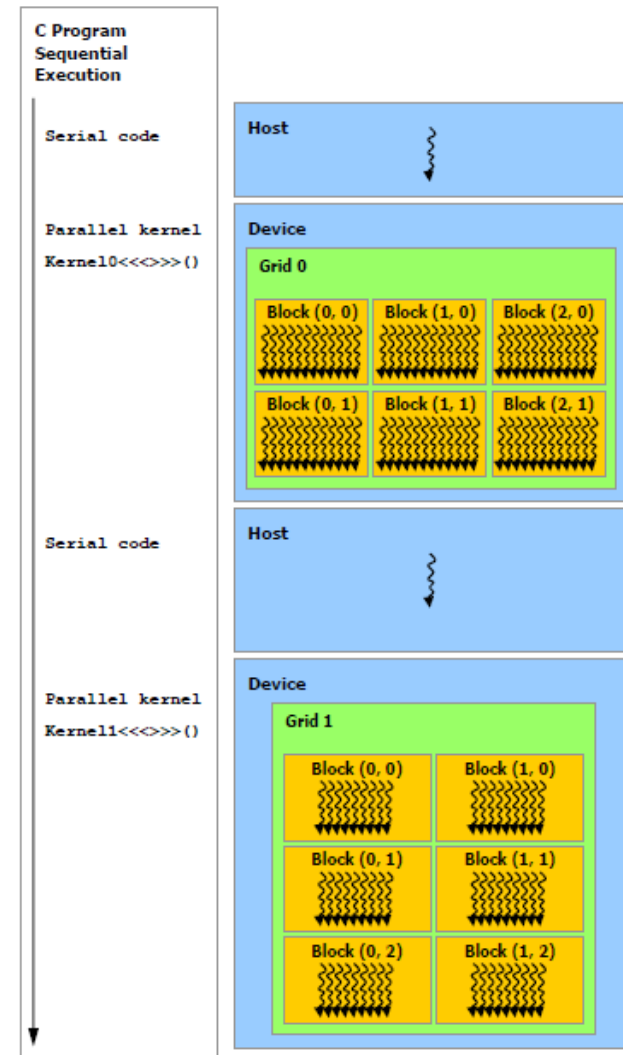


Memory types (fastest memory first):

- Registers
- Shared memory
- Device memory (texture, constant, local, global)



- CUDA assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program.
- The kernels execute on a GPU and the rest of the C program executes on a CPU.
- CUDA assumes that both the host and the device maintain their own separate memory spaces in host and coprocessor memories.



Threads and vectorization

GP-GPUs have included vector instructions for many years. While there is considerable parallel arithmetic available, if the GPU cores are treated as one long vector processor, many well known algorithms from the Cray-2 and T3E eras *just work* and work very well.

This is not a new concept. Early Thinking Machines (CM-1 and CM-2) could be treated as either parallel computers or vector machines using the correct programming style.

This is not to mention visualization, which is exceptionally fast.

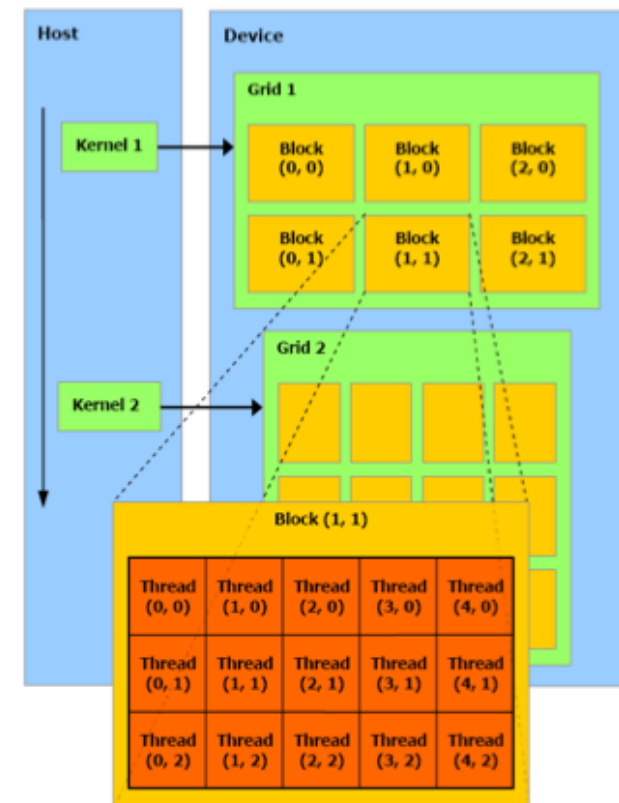
Some examples include

- Cyclic reduction
 - FFT
 - Many relaxation methods
- Multigrid
- Krylov space methods
- PDE discretizations
- Linear algebra (eigencomponent or linear equation solves)
- Nonlinear algorithms
- Time dependent algorithms

Do a vector version where appropriate, do a parallel version where appropriate, or combine both.

Extreme Multithreading

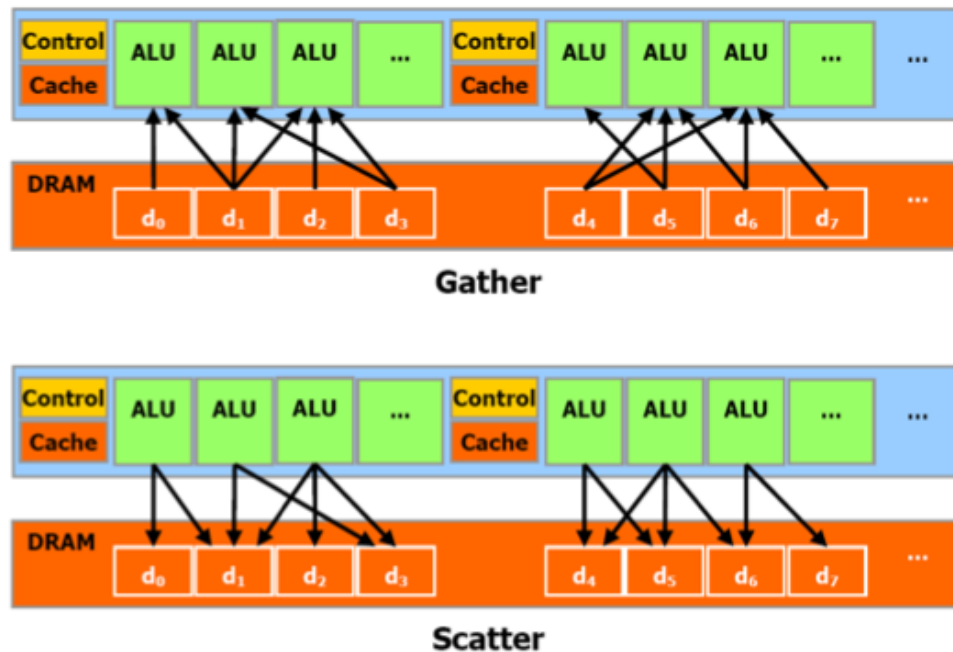
- **Challenge:** 400-600 cycles to get anything from main memory.
- **Solution:** Schedule a vast number of threads to hide latency.
- Grids and thread blocks
 - Organized in 1D/2D/3D blocks/grids
 - Thread blocks are scheduled for execution
- No global thread synchronization, only by thread block.



Data Caching

There is no cohesive memory cache

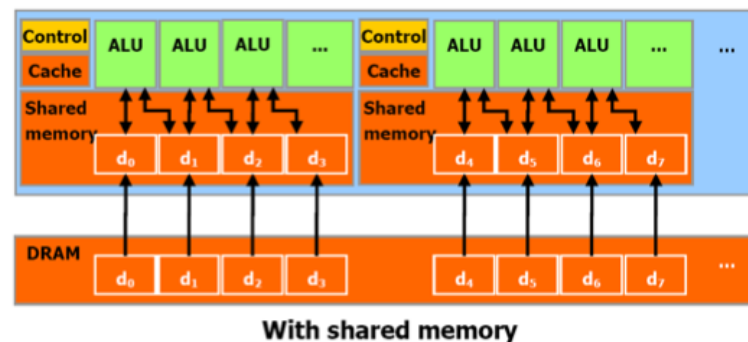
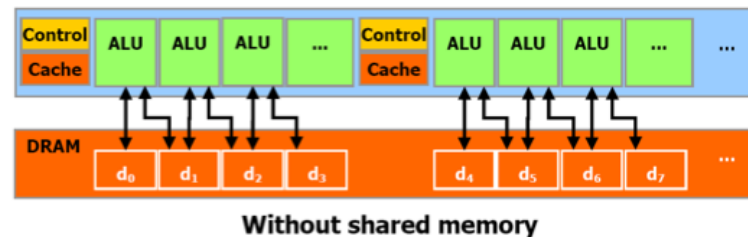
- Scalar processors directly access the on-board DRAM.
- Only a small texture-only cache is available.



Shared Memory

The scalar processors access a relatively small shared memory on the GPU.

Shared memory brings data closer to the ALUs.



Coalesced Memory

Every thread must access the correct memory bank.

On NVIDIA GPUs, 16 memory banks.

- Linear slowdown if using less than the 16 banks in a thread group.
- Correct coalesced memory access:
-



Summary of Implications

Algorithms must fit within all of the restrictions from the preceding slides.

- Very, very bad performance results otherwise.
- Provides a challenge to implementers.
- Design for $O(1000)$ threads per computational kernel
 - Helps if you still remember how to design and program a Thinking Machines CM-1.
- Excellent performance if done right.

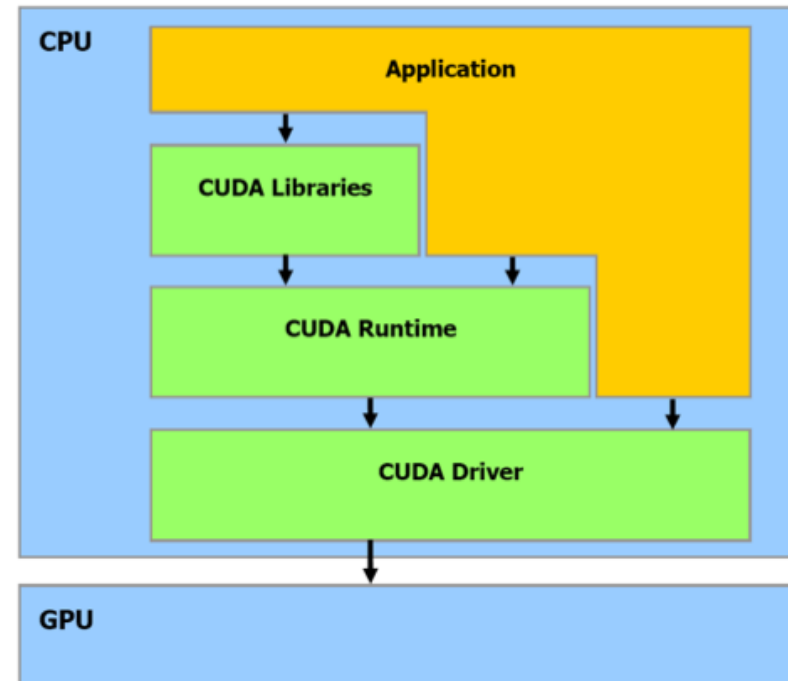
Compilers and Library support

- PGI and Intel compilers have an OpenMP-like pragma set and hide CUDA entirely.
- Nasty new programming methods
 - NVIDIA (CUDA), but OpenCL is even worse.

NVIDIA CUDA Toolkit

C/C++ compilers/libraries

- Library/driver for GPUs
- GPU specific code written separately from CPU code
- Integration with existing compilers
- Encourages incremental GPU coding of existing codes
- Time consuming and memory reusing routines should be coded first in applications
- GPU is a co-processor only



Sparse matrix-vector multiplication

Let A be $N \times N$ stored in compressed row format and u , b N -vectors. Compute $b = Au$. Schedule one thread per row inner product. Sample storage:

4	0	-2	0	0	0	-1	0
-1	4	0	0	0	0	0	-1
0	0	3	-1	0	-1	0	0
0	0	0	4	0	-1	0	0
-1	0	0	0	2	0	-1	0
0	0	-1	0	0	4	0	0
-1	0	0	0	0	3	-1	0
0	0	-1	0	0	-1	0	4

A sample matrix with the rows colored in different hues.

3	3	3	2	3	2	3	3
0	3	6	9	11	14	16	19

1	3	7	1	2	8	3	4	6	4	6	1	5	7	3	6	1	6	7	3	6	8
4	-2	-1	-1	4	-1	3	-1	-1	4	-1	-1	2	-1	-1	4	-1	3	-1	-1	-1	4

Conventional CRS storage (count, displacement, column indices, values)

Looks great, performance terrible.

Convert storage of A into coalesced form by padding with 0's.

Interleave the rows by at least 16 (8 in the example below, however):

3	3	3	2	3	2	3	3
0	1	2	3	4	5	6	7

1	1	3	4	1	3	1	3	3	2	4	6	5	6	6	6	7	8	6		7		7	8
4	-1	3	4	-1	-1	-1	-1	-2	4	-1	-1	2	4	3	-1	-1	-1	-1		-1		-1	4

Use GPU *texture* memory (which is very small in size, unfortunately) for random access of a vector.

Sparse matrix-vector multiplication: Host

```
#define N 256
texture<int2> tex_u;

extern "C" {
void __device_linear_operator(int *cnt, int *dsp, int *col, double *ele,
    int m, int n, double *u, double *v)
{
    cudaBindTexture(0, tex_u, (int2*)u, sizeof(double) * m); //Bind the texture

    struct linear_operator_params parms;
    parms.cnt = cnt;           //ICRS count vector
    parms.dsp = dsp;           //ICRS displacement vector
    parms.col = col;           //ICRS column indices
    parms.ele = ele;           //ICRS matrix entries
    parms.u = u;               //Input vector
    parms.v = v;               //Output vector
    parms.n = n;               //Matrix dimension

    __device_linear_operator<<< (n + N - 1)/N, N >>>(parms); //GPU kernel launch

    cudaUnbindTexture(tex_u); //Unbind the texture
}
}
```

Sparse matrix-vector multiplication: GPU

```
#define L 256

__global__ void __device_linear_operator(struct linear_operator_params parms)
{
    unsigned int j = N * blockIdx.x + threadIdx.x;
    if(j < parms.n)
    {
        unsigned int blkStart = parms.dsp[j];
        unsigned int blkStop = blkStart + L * parms.cnt[j];

        double s = 0.0;
        for(unsigned int i = blkStart; i < blkStop; i += L)    // Interleave stride L
        {
            unsigned int q = parms.col[i];                    //Load column index
            double a = parms.ele[i];                          //Load matrix entry
            int2 c = tex1Dfetch(tex_u, q);                    //Load vector entry from texture cache
            double b = __hiloint2double(c.y, c.x);            //Convert texture entries to double
            s += a * b;                                        //Calculate the sparse scalar product
        }
        parms.v[j] = s;                                       //Store the sparse scalar product
    }
}
```

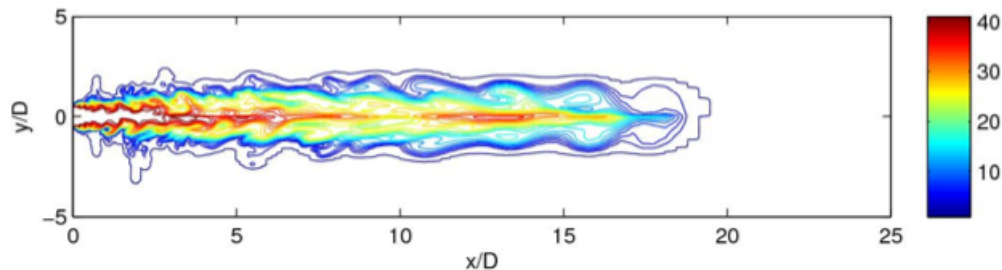
Sparse matrix-vector multiplication: Performance

Finite element fluid dynamics simulation

- (a) $N = 720,000$, $\text{nz}(A) = 5,020,800$
- (b) $N = 274,625$, $\text{nz}(A) = 7,189,057$

Performance in Gigaflops

	AMD Opteron 8347	Intel C2D E6850	Nvidia Geforce 8800 GT
(a)	0.25	0.97	10.1
(b)	0.58	1.17	10.8



Tridiagonal solvers

A tridiagonal system having (n-1)-unknowns can be written in the following form:

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-2} \\ 0 & & & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \end{bmatrix}.$$

Traditional linear time algorithm (Thomas)

Forward sweep:

$$c'_1 = \frac{c_1}{b_1}, \quad d'_1 = \frac{d_1}{b_1},$$
$$c'_i = \frac{c_i}{b_i - c'_{i-1}a_i}, \quad d'_i = \frac{d_i - d'_{i-1}a_i}{b_i - c'_{i-1}a_i}, \quad \text{for } i = 2, \dots, n-2$$

Backward substitution:

$$x_n = d'_n, x_i = d'_i - c'_i x_{i+1} \quad \text{for } i = n-1, \dots, 1$$

The following reduction is possible:

$$a_{i-1}x_{i-2} + b_{i-1}x_{i-1} + c_{i-1}x_i = d_{i-1}$$

$$a_ix_{i-1} + b_ix_i + c_ix_{i+1} = d_i$$

$$a_{i+1}x_i + b_{i+1}x_{i+1} + c_{i+1}x_{i+2} = d_{i+1}$$

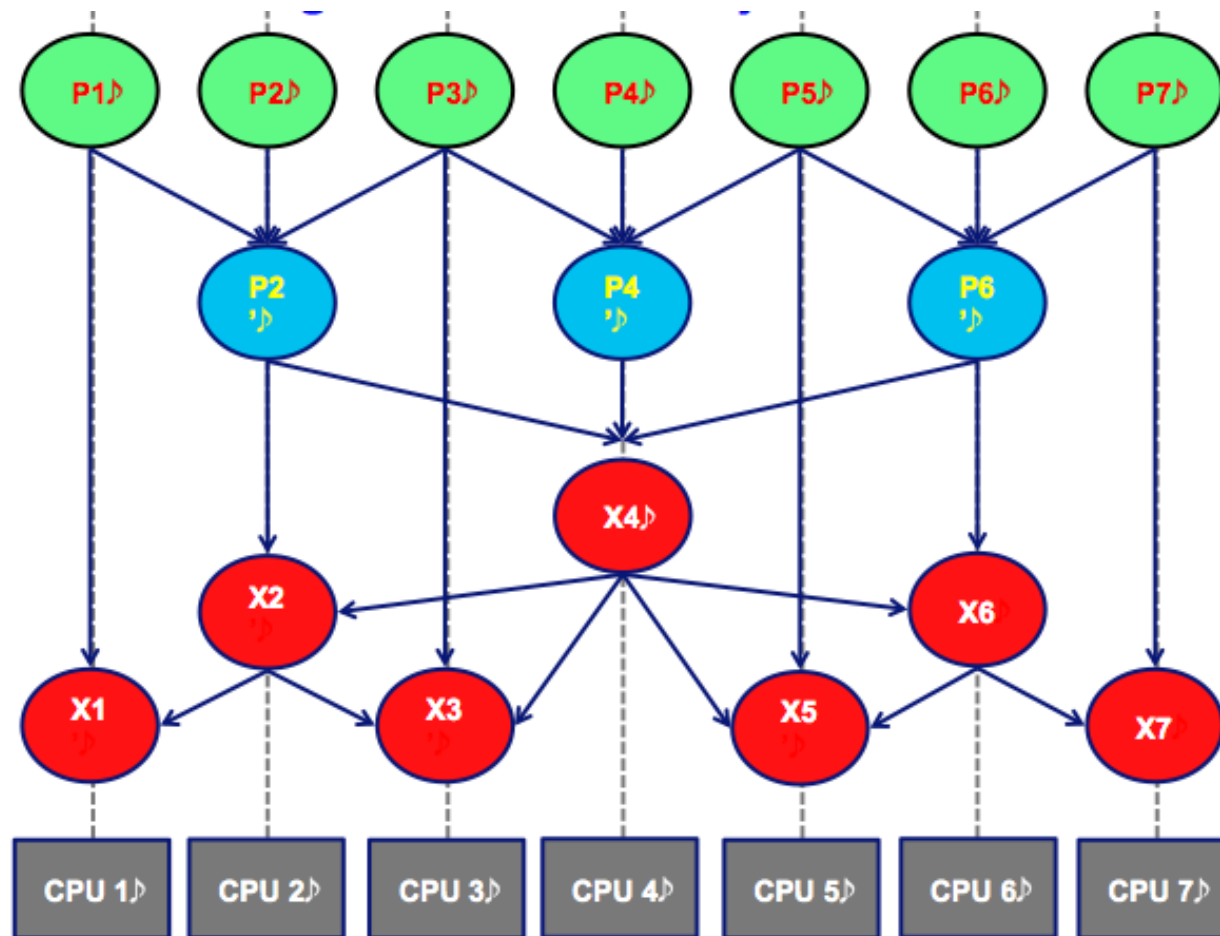
$$a'_ix_{i-2} + b'_ix_i + c'_ix_{i+2} = d'_i$$

$$\text{where } a'_i = -a_{i-1}\alpha, \quad b'_i = b_i - c_{i-1}\alpha - a_{i+1}\beta,$$

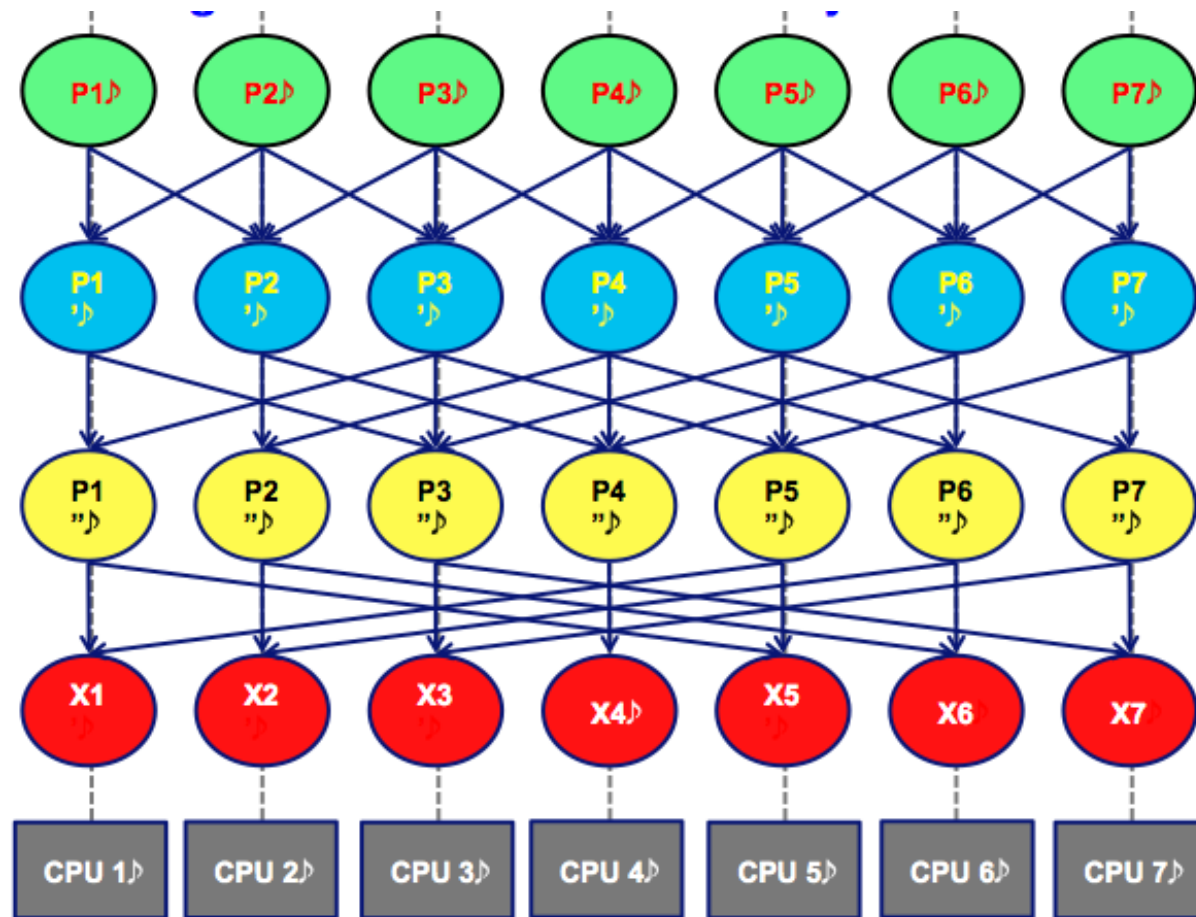
$$c'_i = -c_{i+1}\beta, \quad d'_i = d_i - d_{i-1}\alpha - d_{i+1}\beta,$$

$$\alpha = \frac{a_i}{b_{i-1}}, \text{ and } \beta = \frac{c_i}{b_{i+1}}.$$

Tridiagonal solvers: Cyclic Reduction (CR)



Tridiagonal solvers: Parallel Cyclic Reduction (PCR)



Complexity for n equations:

Method	Arithmetic Operations	Algorithm Steps
Thomas	$8n$	$2n$
CR	$17n$	$2\log_2(n)-1$
PCR	$12n\log_2(n)$	$\log_2(n)$

Scaled computational time:

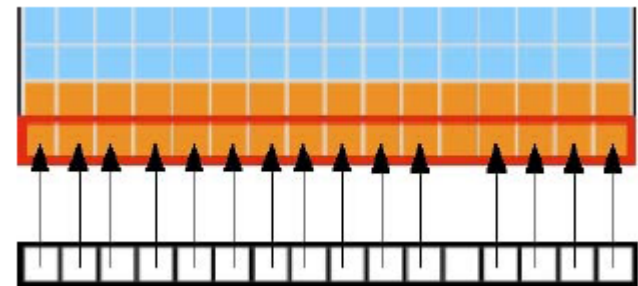
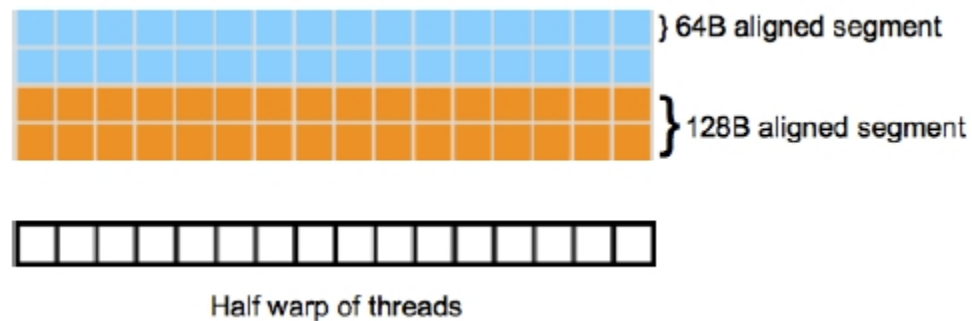
Method	Arithmetic Operations	Algorithm Steps
Thomas	$8n$	$2n$
CR	$17n$	$2\log_2(n)-1$
PCR	$12n\log_2(n)$	$\log_2(n)$

Coalescent memory access (again)

Coalescing over a half wrap of threads (16 threads)

Extreme performance penalty for non-coalescent access

Overcome by a change in both the access pattern and usage of the shared memory



Memory access patterns and performance

Case 1: Assign each continuous four jobs to one thread



Case 2) Assign each continuous five jobs to five threads



Scaled Performance comparison

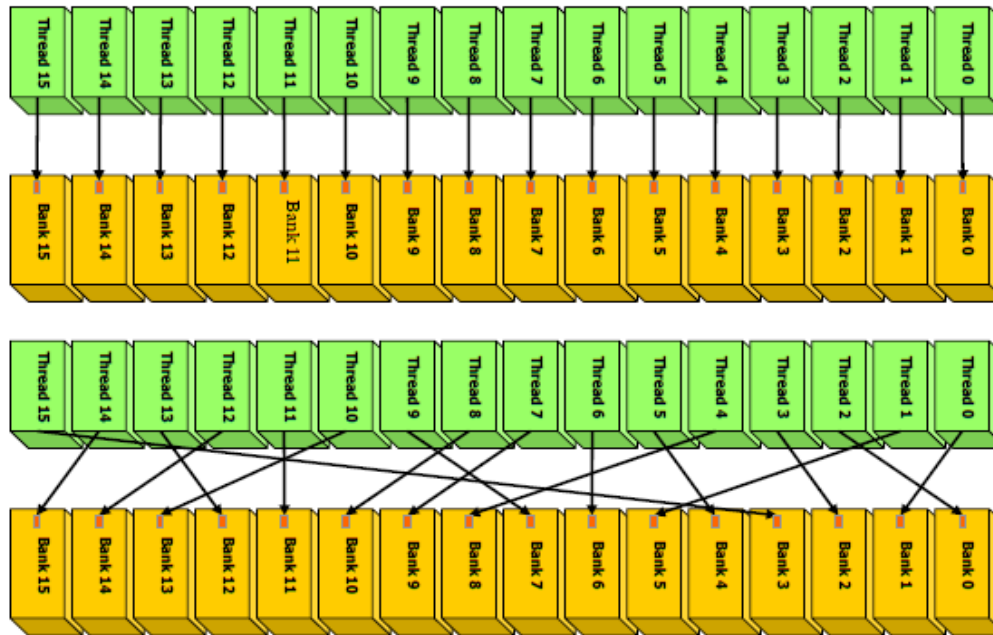
Method	Case 1	Case 2
128x128	1.00	0.62
256x256	1.00	0.41
512x512	1.00	0.38

Memory bank conflicts

Successive 32-bit words are assigned to successive banks

The number of banks is 16

Bank conflicts result in serializing accesses to shared memory



Scaled Computational Time for Double Precision

Method	Thomas		PCR	
Precision	single	double	single	double
64x64	1.00	1.23	0.231	0.692
128x128	1.00	1.69	0.098	0.529
256x256	1.00	1.21	0.044	0.319

Algebraic Multigrid (AMG) on GPU

Advanced solver components

- CPU/GPU parallel AMG component
- Integration with Parallel Toolkit framework
- Support for multi-GPU computing nodes

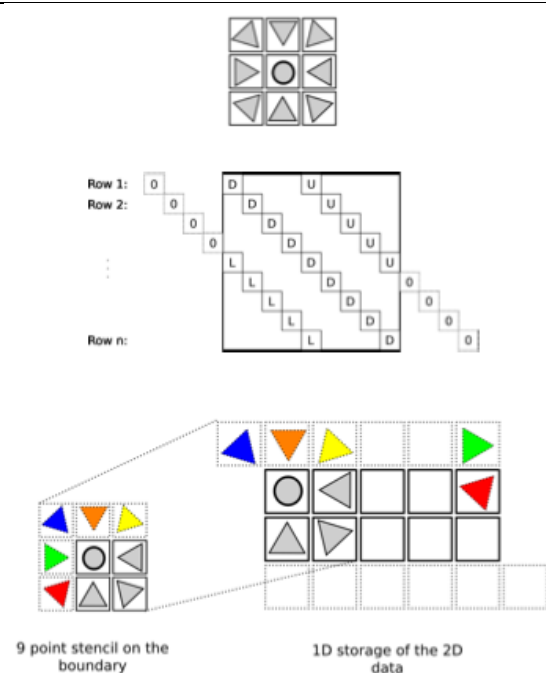
Key performance indicator: Sparse matrix-vector multiplication

- Multigrid prolongation, restriction, and smoother
- Vector operations work well
- Naïve Gauss-Seidel does not work
- High algorithmic complexity in AMG setup phase, which makes it CPU centric

Jacobi Iteration for 2D Structured Mesh

- Bidomain equations on 2D slice of domain
- Discretization with 2D tensor product mesh resulting in 9-pt stencil matrix with variable coefficients.
- Matrix stored as 9 padded diagonals.
- ω -Jacobi for $I=1,2,\dots$:

$$\mathbf{x}_i = \bar{\mathbf{x}}_{i-1}(1 - \omega) + \omega \tilde{\mathbf{D}}^{-1}(\mathbf{f} - (\tilde{\mathbf{L}} + \tilde{\mathbf{U}})\mathbf{x}_{i-1})$$
- Access pattern similar to interpolation in geometric MG.
- 2D vector data stored as 1D vector with additional offset to vectorize matrix-vector product.



Jacobi Iteration on GPU

Avoid uncoalesced memory access by using shared memory.

```
/// allocate shared memory
__shared__ float xl[BLOCKDIMX + 2];
__shared__ float xd[BLOCKDIMX + 2];
__shared__ float xu[BLOCKDIMX + 2];

const int iloc = threadIdx.x + 1;

/// copy data from global GPU memory
/// into shared memory in the block
xl[iloc] = x[tidx-px];
xd[iloc] = x[tidx];
xu[iloc] = x[tidx+px];

if (threadIdx.x == 0){
    xl[0] = x[tidx-px-1];
    xd[0] = x[tidx-1];
    xu[0] = x[tidx+px-1];
}
if (threadIdx.x == blockDim.x-1){
    xl[blockDim.x+1] = x[tidx-px+1];
    xd[blockDim.x+1] = x[tidx+1];
    xu[blockDim.x+1] = x[tidx+px+1];
}
__syncthreads();
```

Jacobi Iteration on GPU Using Textures

Leads to ~32 times faster than on a typical CPU.

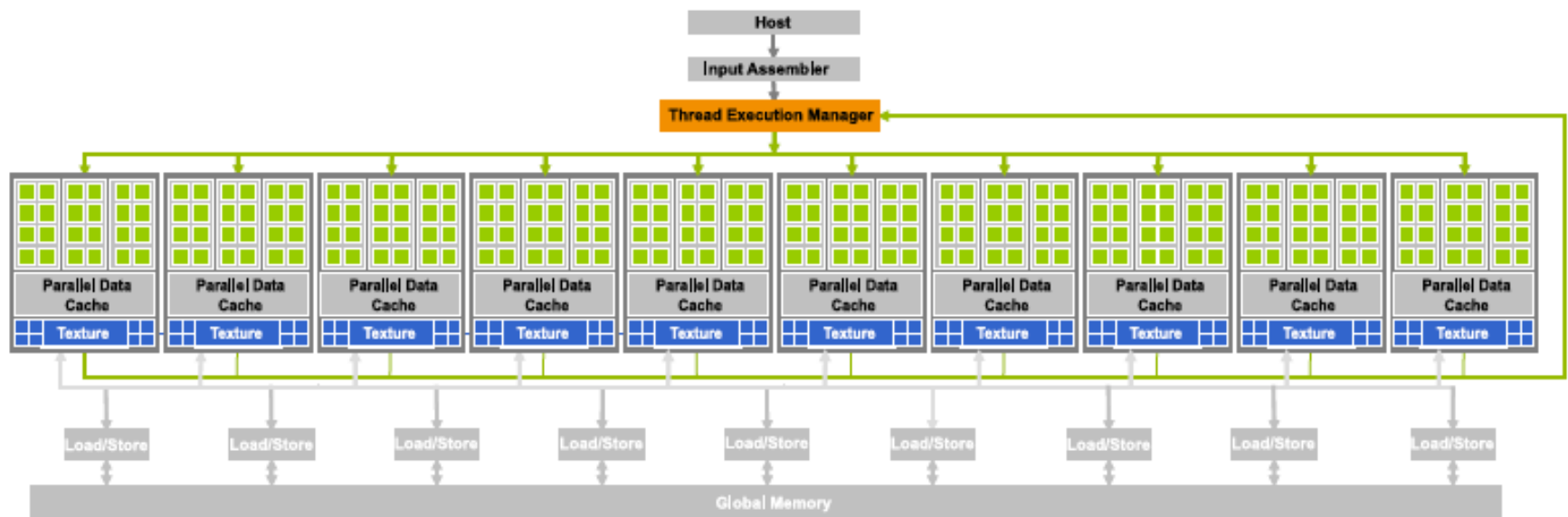
```
texture <float, 1, cudaReadModeElementType> xtex;

__global__ void wjacobi(float *diag0_4, float *diag0_3, float *diag0_2,
                       float *diag0_1, float *diag0_0, float *diag0_5,
                       float *diag0_6, float *diag0_7, float *diag0_8,
                       float *x_out, float *f, float weight, int numits, int nx, int ny)
{
    const int i = blockDim.x*blockIdx.x + threadIdx.x;
    const int N = nx*ny;

    if(i<N){
        x_out[i] = tex1Dfetch(xtex, i)*(1.0-weight) +
            weight*(f[i]-(tex1Dfetch(xtex, i-nx)*diag0_3[i]+
                tex1Dfetch(xtex, i+nx)*diag0_7[i]+
                tex1Dfetch(xtex, i-1)*diag0_1[i]+
                tex1Dfetch(xtex, i+1)*diag0_5[i]+
                tex1Dfetch(xtex, i-nx-1)*diag0_4[i]+
                tex1Dfetch(xtex, i-nx+1)*diag0_2[i]+
                tex1Dfetch(xtex, i+nx-1)*diag0_6[i]+
                tex1Dfetch(xtex, i+nx+1)*diag0_8[i]))/diag0_0[i];
    }
}
```

Tesla Architecture

- ▶ 30 cores, 240 ALUs (1 mul-add)
- ▶ (1 mul-add + 1 mul): $240 * (2+1) * 1.3 \text{ GHz} = 936 \text{ GFLOPS}$
- ▶ 4.0 GB GDDR3, 102 GB/s Mem BW, 4GB/s PCIe BW to CPU



Rules of Hardware resource

- CUDA Threads are not CPU Threads – they are the basic unit of data to be processed
- You can have 64 - 512 Threads per Block
- Grids are made from Blocks and are 1-, 2-, or 3-D
- Threads can share memory with the other Threads in the same Block
- Threads can synchronize with other Threads in the same Block
- Global, Constant, and Texture memory is accessible by all Threads in all Blocks
- Each Thread has registers and local memory
- Each Block can use at most 8,192 registers, divided equally among all Threads

- You can be executing up to 8 Blocks and 768 Threads simultaneously per multiprocessor (MP)
- A Block is run on only one MP (i.e., cannot switch to another MP)
- A Block can be run on any of the 8 processors of its MP
- A Warp is 32 Threads, which is the minimum to run in SIMD fashion

Function qualifiers

`__global__`, `__device__`, `__host__`

These identify where the function runs, e.g.,

```
__global__ void wuf( int* n, char* d ) { ... }
```

Restrictions:

- `__host__` is the default
- No function pointers
- No recursion
- No static variables
- No variable number of arguments
- No return value

Variable qualifiers

`__device__`, `__constant__`, `__shared__`

Examples:

```
__device__ float m[20][10];  
__shared__ int n[32];
```

Default:

- `__device__` in registers

Built-in variables

Available inside kernel code are

- Thread index (type `dim3`): `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
- Block index within grid (type `dim3`): `blockIdx.x`, `blockIdx.y`
- Dimension of grid (type `dim3`): `gridDim.x`, `gridDim.y`
- Dimension of block (type `dim3`): `blockDim.x`, `blockDim.y`, `blockDim.z`
- Warp size: `warpSize`

Intrinsic functions

```
void __syncthreads();
```

Use this to synchronize all threads in the current block. However, it can cause deadlocks, so use with care.

```
__sinf( float ), __cosf( float ), __expf( float )
```

and the usual math intrinsic functions. Beware of `sin(double)` and similar double precision intrinsics that run very slowly in comparison to single precision intrinsics.

Function calls

```
__global__ void wuf( int* n, char* d );  
...  
dim3 grid(16,16);  
dim3 block(16,16);  
wuf <<< grid,block,0,0 >>>(n, d);  
wuf <<< grid,block >>>(n, d);
```

Launch parameters

- Grid dimension (1- or 2-D)
- Block dimension (1-, 2-, or 3-D)
- Optional stream id and shared memory size
- `kernel<<< grid, block, stream, shared_mem >>> ();`

Memory management

Host manages GPU memory

- `cudaMalloc(void** ptr, size_t size);`
- `cudaMemset(void* ptr, int value, size_t count);`
- `cudaFree(void* ptr);`

Memcpy for GPU

- `cudaMemcpy(void* dst, void* src, size_t size, cudaMemcpyKind);`

`cudaMemcpyKind`

- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`

Time measurement on GPU

Do not measure launch time as part of running time.

```
int timer=0;  
cutCreateTimer( &timer );  
cutStartTimer( timer );  
...  
cutStopTimer( timer );  
cutGetTimerValue( timer );  
cutDeleteTimer( timer );
```

Use events for asynchronous functions

```
cudaEvent_t start_event, stop_event;  
cutilSafeCall( cudaSafeEventCreate( &start_event ) );  
cutilSafeCall( cudaSafeEventCreate( &stop_event ) );  
// Record in stream 0 that all previous CUDA calls are done  
cudaEventRecord( &start_event, 0 );  
...  
cudaEventRecord( &stop_event, 0 );  
// Block until the event is actually recorded  
cudaEventSynchronize( stop_event );  
cudaElapsedTime( &time_memcpy, start_event, stop_event );
```

Useful code to choose the fastest GPU on your system

```
int num_devices, device;
cudaGetDeviceCount(&num_devices);
if (num_devices > 1) {
    int max_multiprocessors = 0, max_device = 0;
    for (device = 0; device < num_devices; device++) {
        cudaDeviceProp properties;
        cudaGetDeviceProperties(&properties, device);
        if (max_multiprocessors < properties.multiProcessorCount){
            max_multiprocessors = properties.multiProcessorCount;
            max_device = device;
        }
    }
    cudaSetDevice(max_device);
}
```

Libraries

- CUBLAS
- CUFFT
- CULA
- CULAPACK

Debuggers

- cuda-gdb
- visual debugger

Compilers

- Now based on LVMM project

Good sources of information

- NVIDIA Developer Zone
- NVIDIA Webinar,
http://developer.download.nvidia.com/CUDA/training/cuda41_webinar.mp4
- http://geco.mines.edu/tesla/cuda_tutorial_mio/index.html
- <http://pdsgroup.hpclab.ceid.upatras.gr/files/CUDA-Parallel-Programming-Tutorial.pdf>
- Dr. Dobbs journal, CUDA, Supercomputing for the Masses, parts 1-18