

# An introduction to heterogenous computing

## Accelerating numerical codes

Krishna Kumar \*<sup>1</sup>

August 11, 2015

---

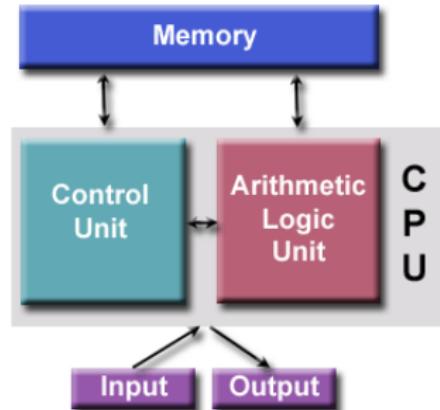
<sup>1</sup>[github.com/kks32](https://github.com/kks32)

# Outline

- 1 Overview: Architecture
- 2 Parallel computing
- 3 Shared memory
- 4 Distributed memory
- 5 Graphics Processing Units

# von Neumann Architecture

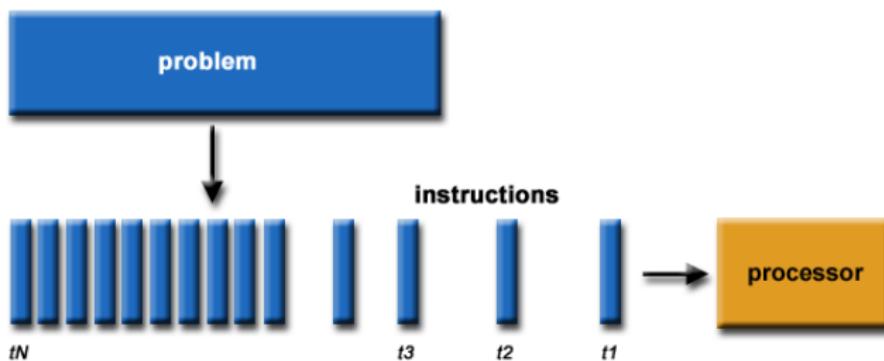
- Hungarian mathematician John von Neumann circa 1940 - the general requirements for an electronic computer.
- “Stored-program computer” - both program instructions and data are kept in electronic memory.
  - *Read/write*, random access memory is used to store both program instructions and data.
  - *Control unit* fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.
  - *Arithmetic Unit* performs basic arithmetic operations
  - *Input/Output* is the interface to the human operator



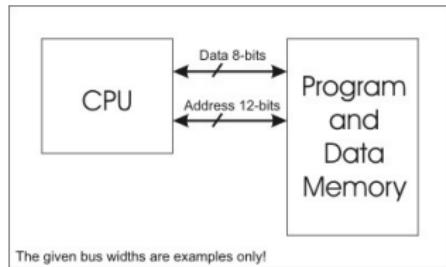
Basic architecture

# Serial computing

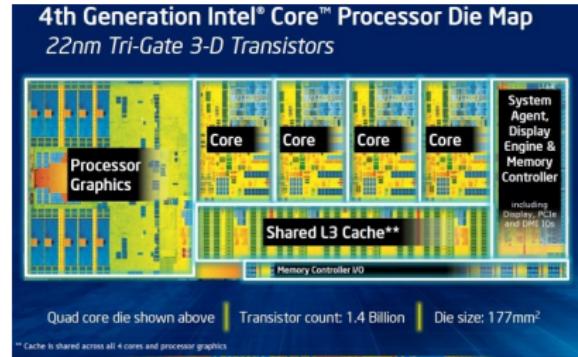
- Traditionally, software has been written for serial computation:
  - A problem is broken into a discrete series of instructions
  - Instructions are executed sequentially one after another
  - Executed on a single processor
  - Only one instruction may execute at any moment in time



# Memory model



Ideal memory model:  
We write for this architecture



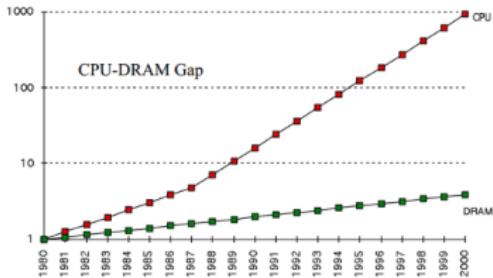
Real memory model: How it actually looks

*The underlying assumption is cache coherency!*

In a shared memory multiprocessor with a separate cache memory for each processor, it is possible to have many copies of any one instruction operand: one copy in the main memory and one in each cache memory. When one copy of an operand is changed, the other copies of the operand must be changed also. Cache coherency ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion.

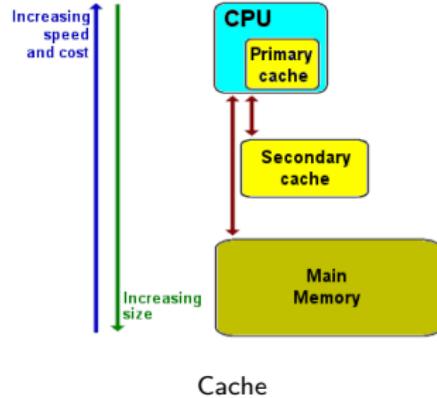
# What are caches

- Processor vs Memory Performance



1980: no cache in microprocessor;  
1995 2-level cache

CPU vs DRAM



- CPU caches are small pools of memory that store information the CPU is most likely to need next.
- A cache miss means the CPU has to go scampering off to find the data elsewhere. This is where the L2 cache comes into play while it's slower, it's also much larger.
- If data can't be found in the L2 cache, the CPU continues down the chain to L3 (typically still on-die), then L4 (if it exists) and main memory (DRAM).

# Does your compiler execute the program you wrote?

**No, absolutely not! Compiler most often says “you didn’t intend to write that. I have a better idea...”**

- *Sequential consistency*: Executing the program you wrote.  
“the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” - Lesslie Lamport
- *Compiler optimisation*
- *Processor execution*
- *Cache coherency*
- Chip / compiler design annoyingly helpful:
  - It can be expensive to exactly execute what you wrote
  - Often they rather do something else, that's faster

# Does your compiler execute the program you wrote?

---

```
// Your code
1 for (i = 0; i < rows; ++i) {
2     for (j = 0; j < cols; ++j) {
3         a[j*rows+i]+=42;
4     }
5 }
6 }
```

---

```
// Compiler optimised version
1 for (j = 0; j < cols; ++j) {
2     for (i = 0; i < rows; ++i) {
3         a[j*rows+i]+=42;
4     }
5 }
6 }
```

---

The CPU will expect a sequential operation. Iterating through each row of data is faster than going through each column. Almost always, a 2D matrix is stored as a 1D linear array.

# Stack v Heap

- **Stack:** Stores local data, return addresses, used for parameter passing. e.g., int i;
- typically stored in the “low” addresses of memory and fills upward toward its upper limit.
- faster, but smaller in size. Last In First Out.
- **Heap:** You would use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.  
*ClassObj \* obj = new ClassObj{0}; ... delete obj;  
auto obj = std::make\_shared<ClassObj>();*
- Stored at the “top” of the address space and grows towards the stack.
- Slower but larger in size
- Use local variables (stack) when you can. Use dynamic allocation (heap) when you have to.



Stack v Heap

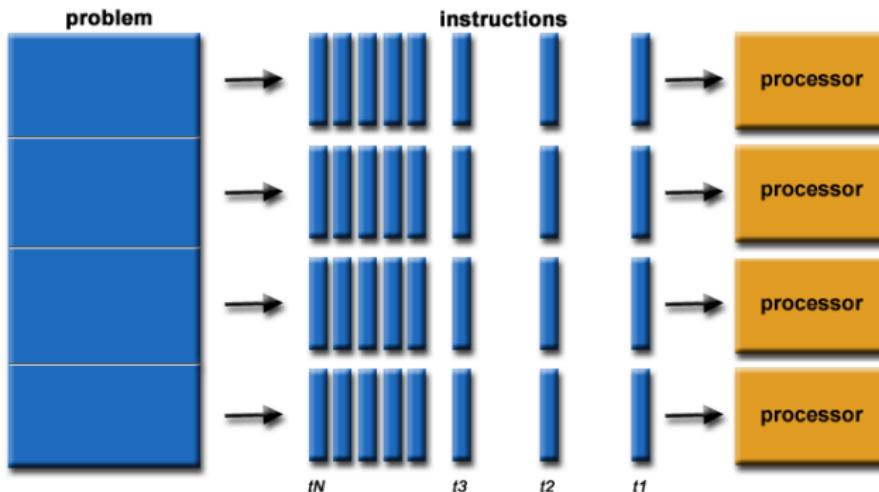
# Outline

- 1 Overview: Architecture
- 2 Parallel computing
- 3 Shared memory
- 4 Distributed memory
- 5 Graphics Processing Units

# What is parallel computing?

In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed



# Concurrency v Parallelism

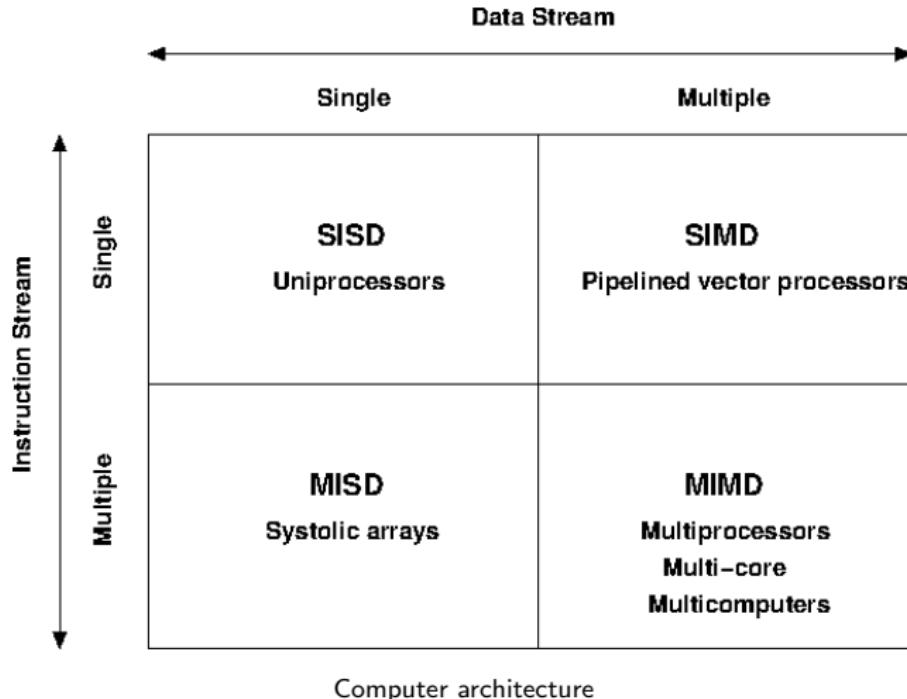


Concurrency



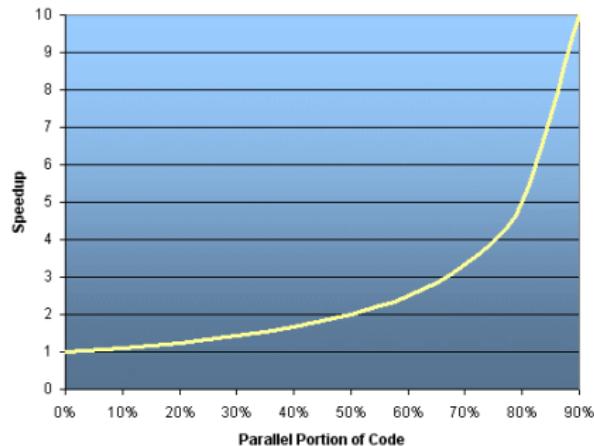
parallelism

# Flynn's Classical Taxonomy



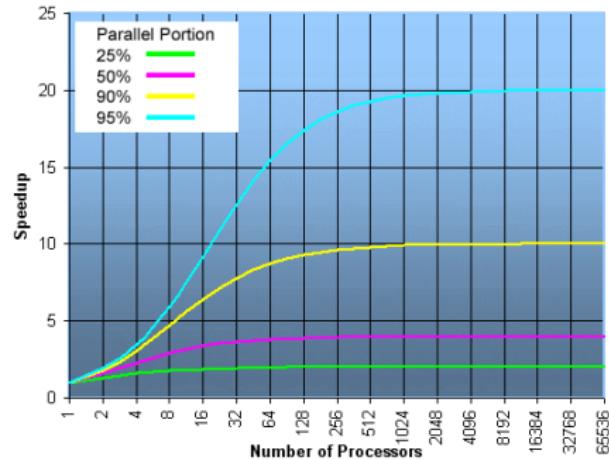
# Cost of parallelisation

$$speedup = \frac{1}{1 - \text{parallelpart}}$$



Single processor

$$speedup = \frac{1}{\frac{\text{parallelpart}}{\#\text{processors}} + \text{serialpart}}$$



Multiple processor

Problems that increase the percentage of parallel time with their size are more **scalable** than problems with a fixed percentage of parallel time

# Scalability

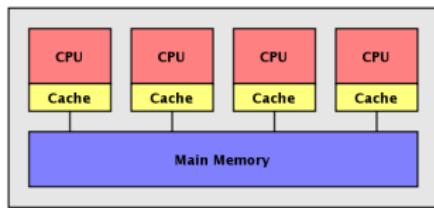
- *Strong scaling:* The total problem size stays fixed as more processors are added.
- *Weak scaling:* The problem size per processor stays fixed as more processors are added.
- Simply adding more processors is rarely the answer to scalability.
- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease.
- Hardware factors play a significant role in scalability. Examples:
  - Memory-cpu bus bandwidth on Symmetric Multi-Processor (SMP) - where multiple processors share a single address space and have equal access to all resources.
  - Communications network bandwidth
  - Amount of memory available on any given machine or set of machines
  - Processor clock speed
- Parallel support libraries and subsystems software can limit scalability independent of your application.

# Outline

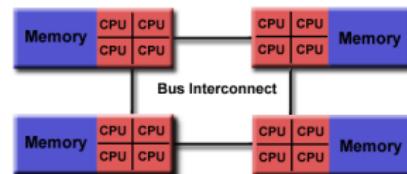
- 1 Overview: Architecture
- 2 Parallel computing
- 3 Shared memory
- 4 Distributed memory
- 5 Graphics Processing Units

# Shared memory

- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs
- Primary disadvantage is the lack of scalability between memory and CPUs

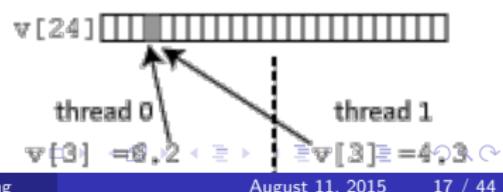


Shared memory



Non Uniform Memory access

If we want thread 0 to use the value placed in the array by thread 1, we need to use a mechanism which assures that thread 1 has written the value before thread 0 reads it.



- **OpenMP**
  - API that supports multi-platform shared memory multiprocessing programming
  - Model: master thread forks
  - Simple to use
- **POSIX threads or p-threads**
  - Library based set of C language types, functions and constants.
  - A thread can be created with much less operating system overhead.
  - No advanced threading algorithm. Use C++11 with same set of features.
- **C++11 threads**
  - C++11 language provides a memory model that supports threading.
  - This library provides everything from thread management to mutexes and synchronization.
- **Intel Cilk**
  - Language extension.
  - Simple to parallelise.
  - No advanced algorithm support.
- **Intel Threaded Building Blocks**
  - C++ template library for parallelisation.
  - Support for parallel algorithms, data structures and is scalable.

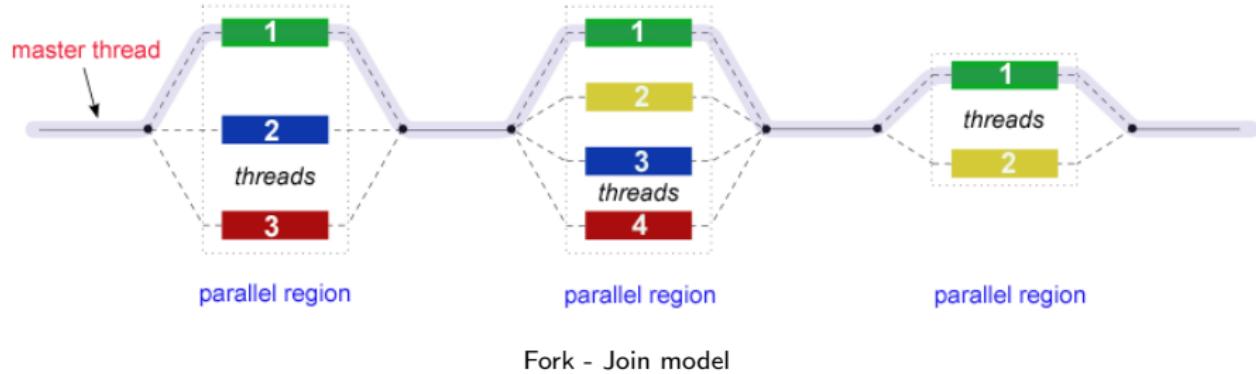
# Shared memory: Choosing the right framework

OpenMP	Intel Cilk Plus	Intel TBB	C++11 Threads	PThreads
+ Options	+ Performance	+ Possibilities	+ Flexibility	+ Flexibility
+ Portable	+ Scaling	+ Management	+ Type-Safety	+ Low-Level
+ Languages	+ Variables	+ Maintenance	+ Possibilities	+ Compatibility
- Performance	- Fortran	- Language	- Fortran	- Efforts
- Memory	- Control	- OOP	- Compiler	- Type-Safety
- Unreliable	- Possibilities	- Control	- Scaling	- Management

# OpenMP

- Open Multi-Processing is an API to explicitly direct multi-threaded, shared memory parallelism
- Comprised of three primary API components:
  - Compiler Directives Pragmas (pre-processor macros)
  - Runtime Library Routines
  - Environment Variables
- The programmer is responsible for synchronizing input and output.
- OpenMP uses threads. A thread of execution is the smallest unit of processing that can be scheduled by an operating system.
- Threads exist within the resources of a single process. Without the process, they cease to exist.
- Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.

# Fork Join Model



- **FORK:** the master thread then creates a team of parallel threads. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.

# OpenMP: Code structure

---

```
1 #include <omp.h>
2 int main () {
3     int var1, var2, var3;
4     // Serial code . . .
5     // Beginning of parallel section. Fork a team of threads.
6     //Specify variable scoping
7 #pragma omp parallel for private(var1, var2) shared(var3) {
8     // Parallel section executed by all threads
9     // Run-time Library calls
10    // All threads join master thread and disband
11 }
12 // Resume serial code ..
13 }
```

---

# OpenMP: Dot product

Dot product  $sum+ = a[N] * b[N]$

---

```
1 // serial code
2 const int size = 100;
3 int main() {
4     int i;
5     float a[size], b[size];
6     float sum = 0.0;
7     // Some initializations
8     for (i = 0; i < size; i++)
9         a[i] = b[i] = 1.0;
10    // computation loop
11    for (i = 0; i < size; i++)
12        sum += (a[i] * b[i]);
13
14    printf("Sum = %f\n", sum);
15 }
```

---

---

```
1 // parallel code
2 const int size = 100;
3 int main() {
4     int i; float sum = 0.0;
5     float a[size], b[size];
6
7     for (i = 0; i < size; i++)
8         a[i] = b[i] = 1.0;
9
10    #pragma omp parallel for \
11        default(shared) private(i)
12    for (i = 0; i < size; i++)
13        sum += (a[i] * b[i]);
14
15    printf("Sum = %f\n", sum);
16 }
```

---

# Parallel dot product: What went wrong?

---

```
1 // wrong code
2 const int size = 100;
3 int main() {
4     int i; float sum = 0.0;
5     float a[size], b[size];
6     for (i = 0; i < size; i++)
7         a[i] = b[i] = 1.0;
8
9 #pragma omp parallel for \
10 default(shared) private(i)
11 for (i = 0; i < size; i++)
12     sum += (a[i] * b[i]);
13
14 printf("Sum = %f\n", sum);
15 }
```

---

```
1 // parallel code
2 const int size = 100;
3 int main() {
4     int i; float sum = 0.0;
5     float a[size], b[size];
6     for (i = 0; i < size; i++)
7         a[i] = b[i] = 1.0;
8
9 #pragma omp parallel for \
10 default(shared) private(i) \
11 reduction(+ : sum)
12 for (i = 0; i < size; i++)
13     sum += (a[i] * b[i]);
14
15 printf("Sum = %f\n", sum);
16 }
```

---

Synchronisation! Don't forget to synchronise the threads

# Auto vectorization

- A special case of automatic parallelisation, where a computer program is converted from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation, which processes one operation on multiple pairs of operands at once.
- The Auto-Vectorizer analyzes loops in your code, and uses the vector registers and instructions on the target computer to execute them, if it can. This can improve the performance of your code.

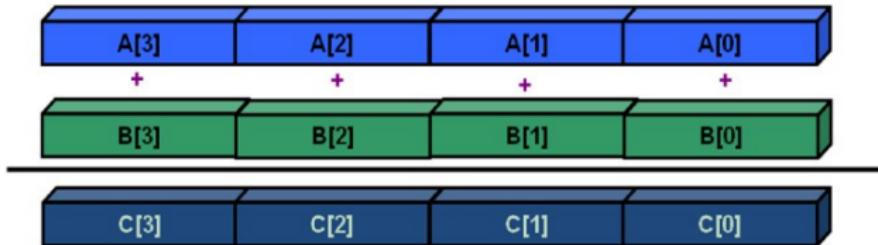
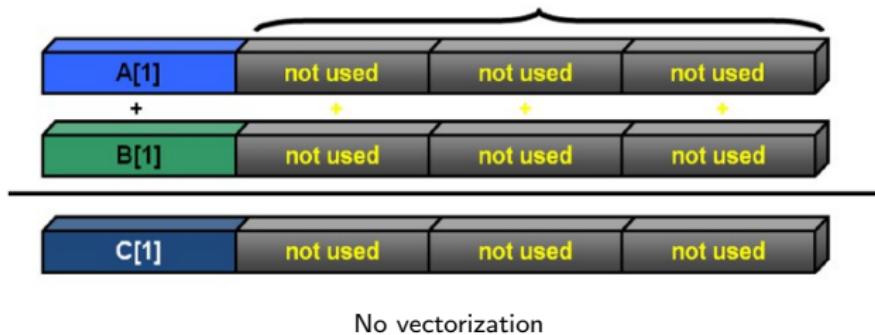
---

```
1 // sequential code                                // vectorized code
2
3 for (int i = 0; i < 1000; ++i) {                2 for (int i = 0; i < 1000; i+=4) {
4   //32-bit operation                            3   A[i:i+3] = mulps(B[i:i+3]*C[i:i+3]);
5   A[i] = B[i]*C[i];                           4   // 128-bit operation
6 }                                              5   // Which is 4x32bit operations
7 }                                              6   // Takes the same amount of time
8 }
```

---

# Auto vectorization: How it works

e.g.  $3 \times 32$



Vectorized version

# SIMD parallelisation in loops

- Loop level parallelism
  - SIMD for a single statement across consecutive iterations
  - Handles misaligned data
  - Patterns such as reduction, linear recursion
- SIMD across entire loop iterations
- Effectively collapse innermost loop

```
for (k=0; k<N; k++) {  
    ...other code...  
    for (i=0; i<8; i++)  
        r[k] += i[k+i] * c[k+i];  
}
```



```
for (k=0; k<N; k++) {  
    ...other code...  
    r[k:k+3] = i[k+i:k+3+i]*c[k+i:k+3+i];  
    r[k+4:k+7] = i[k+4+i:k+7+i]*c[k+4+i:k+7+i];  
}
```

SIMD nested loops

# SIMD parallelisation in loops

```
for (i=0; i<N; i+=4) {  
    a[i] = b[i] + c[i];  
    a[i+1] = b[i+1] + c[i+1];  
    a[i+2] = b[i+2] + c[i+2];  
    a[i+3] = b[i+3] + c[i+3];  
}
```

unrolled  
loop

```
for (i=0; i<N; i+=4) {  
    a[i:i+3] = b[i:i+3] + c[i:i+3];  
}
```

```
for (i=0; i<N; i++) {  
    p = &a[i]; q = &b[i];  
    p.x = q.x + ...  
    p.y = q.y + ...  
    p.z = q.z + ...  
}
```

structure

```
for (i=0; i<N; i++) {  
    p = &a[i]; q = &b[i];  
    p.xyz = q.xyz + ...  
}
```

x	y	z	dummy
---	---	---	-------

```
s += a[i]*b[i] + a[i+1]*b[i+1] +  
     a[i+2]*b[i+2] + a[i+3]*b[i+3];
```

statement

```
t[i:i+3] = a[i:i+3] * b[i:i+3];  
s += t[i]+t[i+1]+t[i+2]+t[i+3];
```

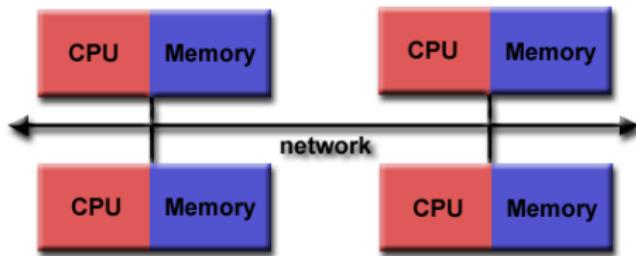
# Outline

- 1 Overview: Architecture
- 2 Parallel computing
- 3 Shared memory
- 4 Distributed memory
- 5 Graphics Processing Units

# Distributed memory

- **General Characteristics:**

- Distributed memory systems require a communication network to connect inter-processor memory. Usually via Ethernet.
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated.



# Distributed memory: Advantages and disadvantages

- **Advantages:**

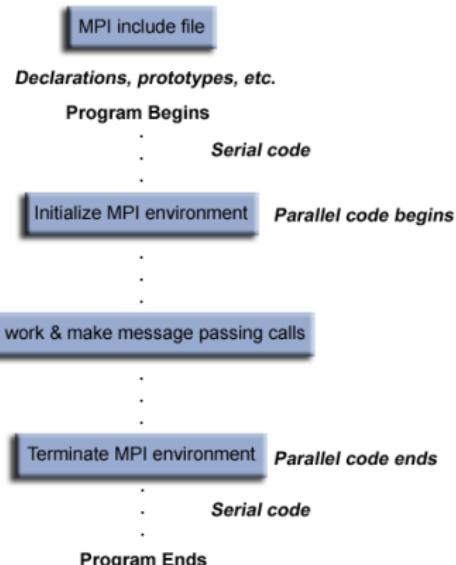
- Memory is scalable with the number of processors.
- Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain global cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

- **Disadvantages:**

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access times - data residing on a remote node takes longer to access than node local data.

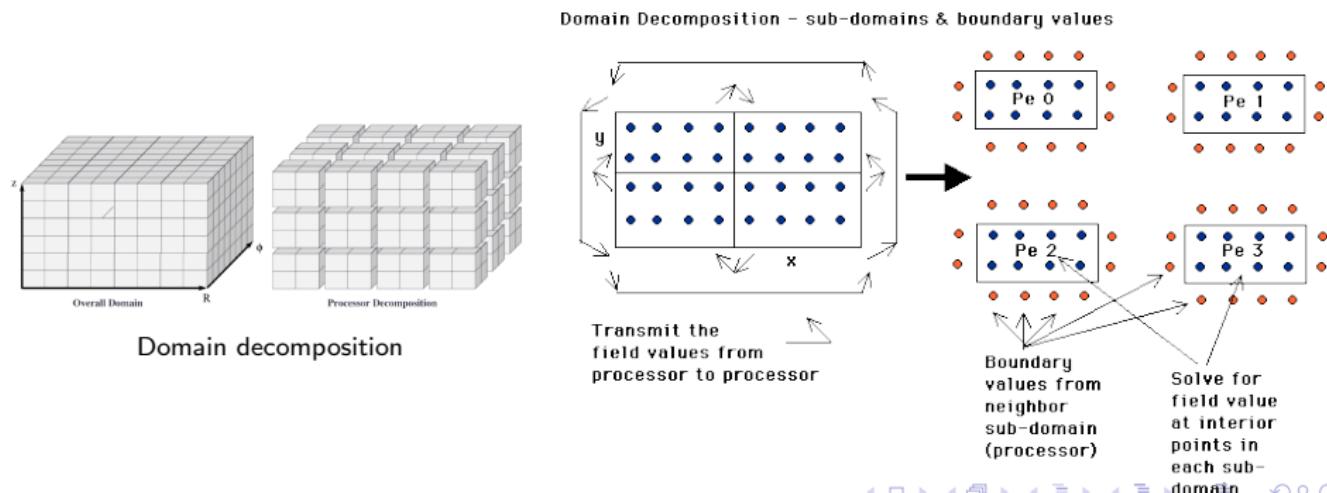
# MPI: Message Passing Interface

- OpenMPI, MPICH, IntelMPI, BoostMPI
- MPI libraries vary in their level of thread support:
  - *MPI\_THREAD\_SINGLE* - Level 0: Only one thread will execute.
  - *MPI\_THREAD\_FUNNELED* - Level 1: The process may be multi-threaded, but only the main thread will make MPI calls - all MPI calls are funneled to the main thread.
  - *MPI\_THREAD\_SERIALIZED* - Level 2: The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time. That is, calls are not made concurrently from two distinct threads as all MPI calls are serialized.
  - *MPI\_THREAD\_MULTIPLE* - Level 3: Multiple threads may call MPI with no restrictions.



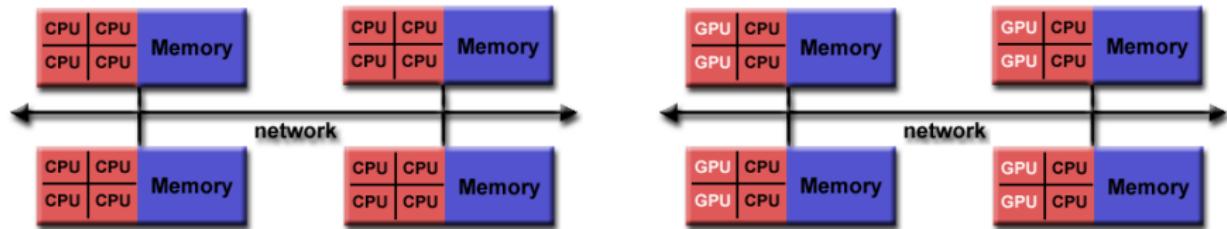
# Domain decomposition

- Tasks are statically or semi-statically mapped onto processes based on spatial location; each task performs similar operations on different data (subdomains).
- Work is interspersed with communication to synchronize the tasks or share data.
- The degree of parallelism increases with problem size, enabling effective use of more processes on larger problems.



# Hybrid Distributed-Shared Memory

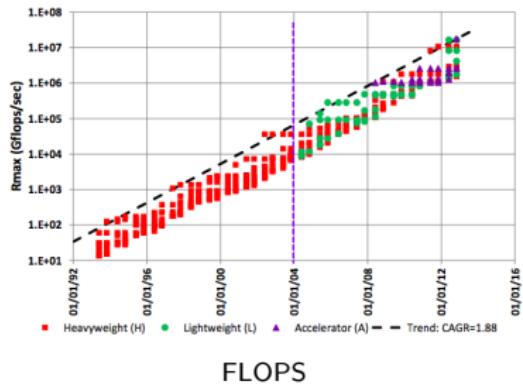
- Largest and fastest computers employ hybrid architectures.
- The shared memory component can be a shared memory machine and/or graphics processing units (GPU).
- The distributed memory component is the networking of multiple shared memory/GPU machines, which know only about their own memory - not the memory on another machine. Therefore, network communications are required to move data from one machine to another.
- **Advantages and Disadvantages:**
  - shared and distributed memory architectures.
  - Increased scalability is an important advantage
  - Increased programmer complexity



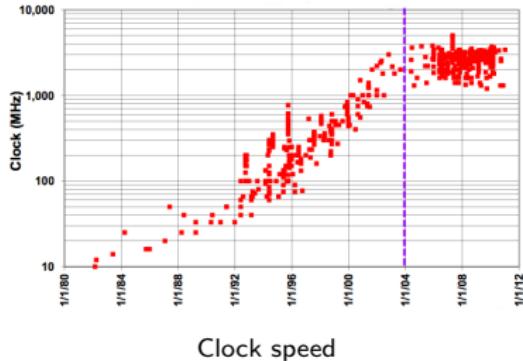
# Outline

- 1 Overview: Architecture
- 2 Parallel computing
- 3 Shared memory
- 4 Distributed memory
- 5 Graphics Processing Units

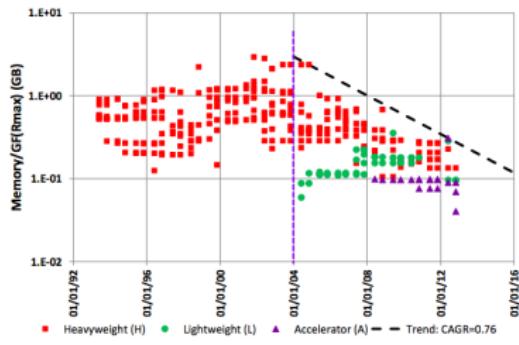
# Trends in HPC



FLOPS



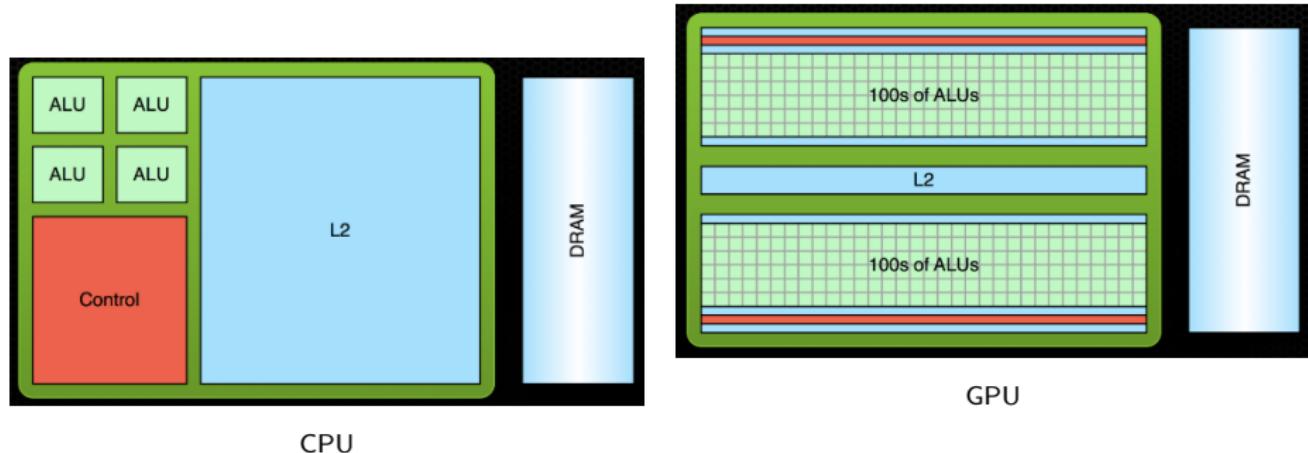
Clock speed



Memory / flop

More Flop/s per clock cycle and less memory per Flop/s means Flop/s are “free” but there is a price: complexity in programmability!

# Graphics Processing Units (GPU)

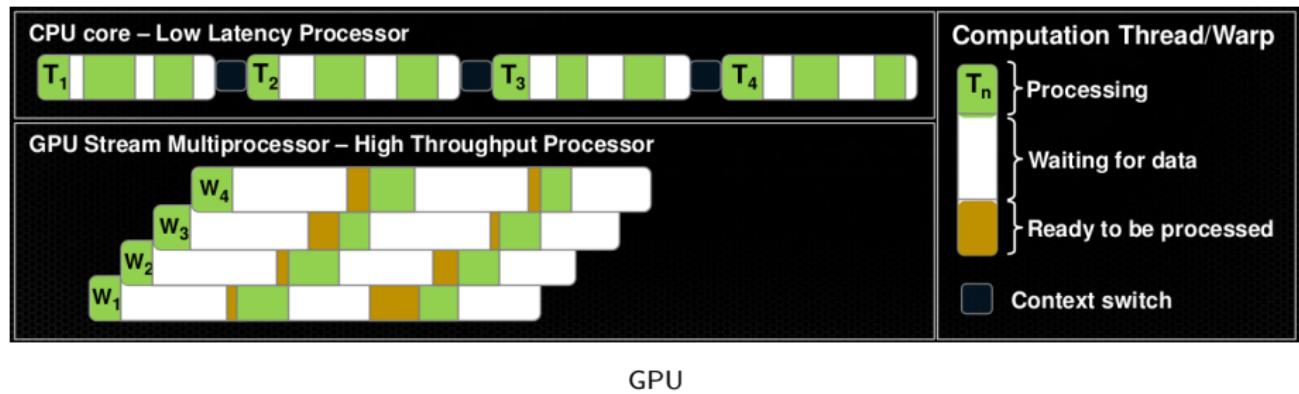


- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution
- 10s of threads

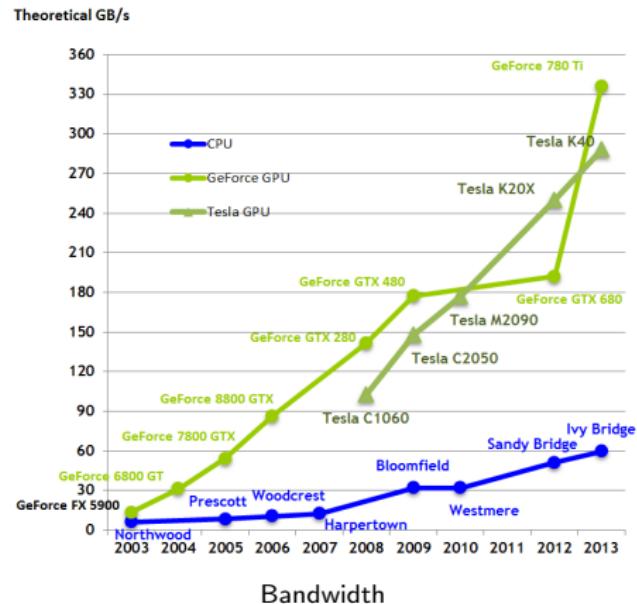
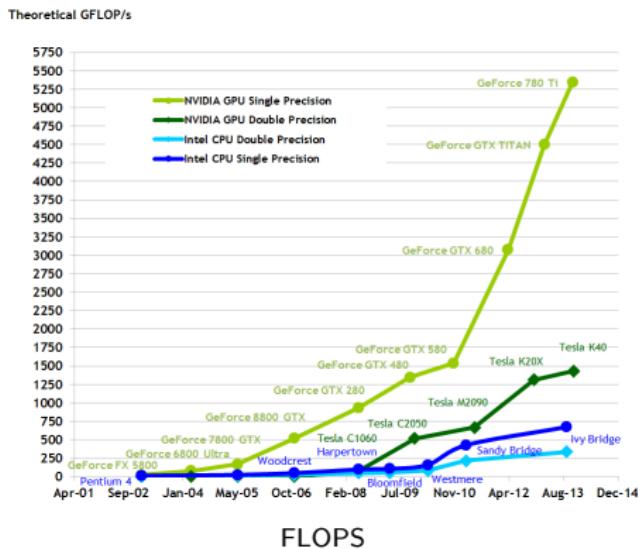
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
- 10000s of threads

# Low Latency or High Throughput?

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps

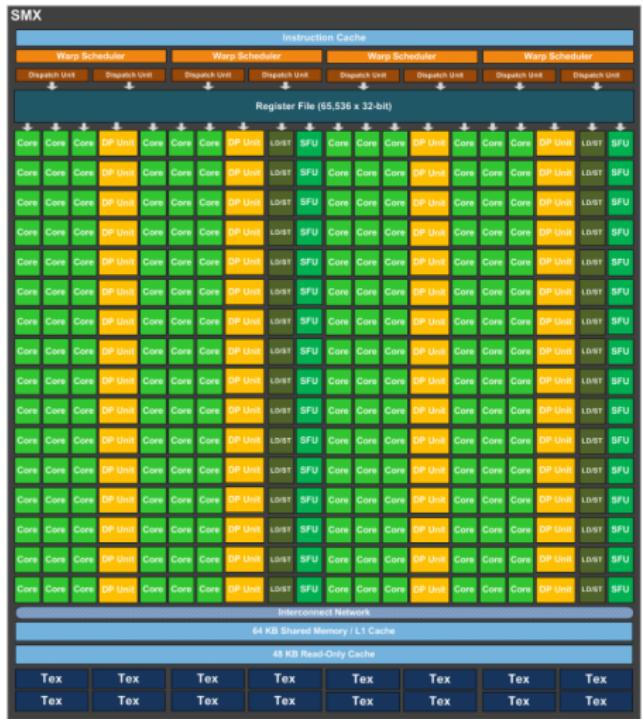


# CPU v GPU



# NVIDIA K20 GPU architecture

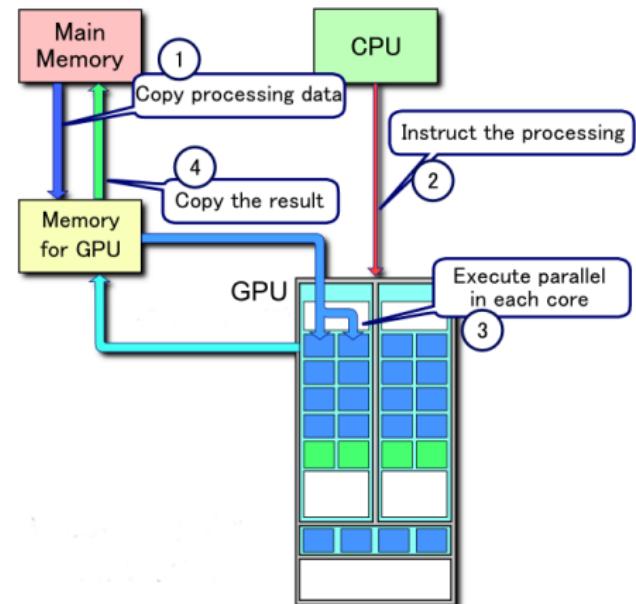
- SMX = Streaming multi processor
  - 192 cores
  - 64k registers
  - 64KB of shared memory and L1 cache
  - 8KB cache for constants
  - up to 2K active threads
- Each K20 has 13 SMX...
  - 200 GB/s
  - 1.1 TFlops (peak)
  - 225 Watt (peak)



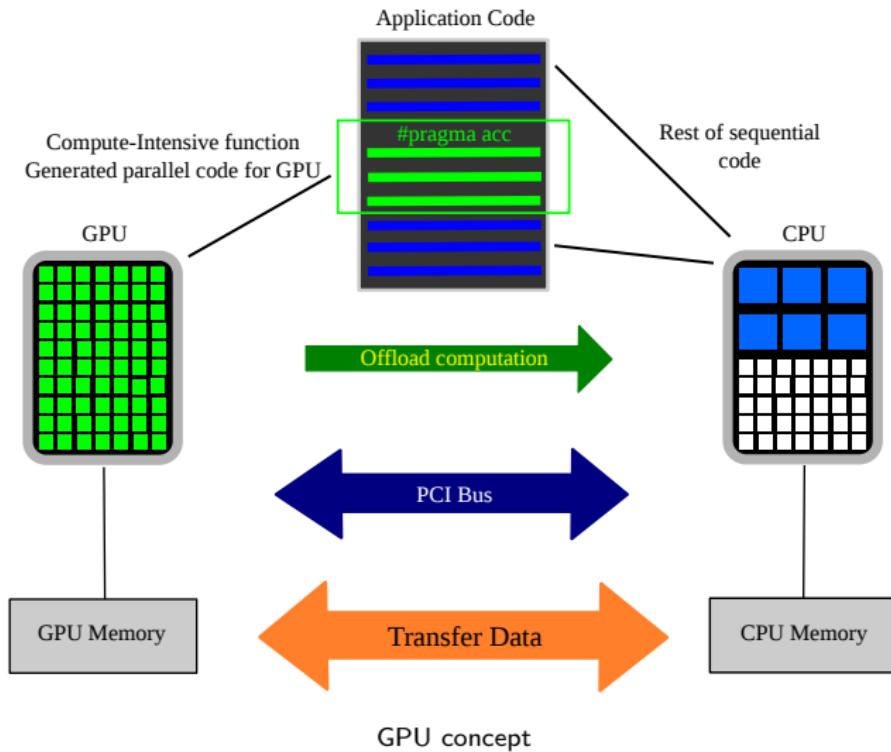
Bandwidth

# GPU workflow

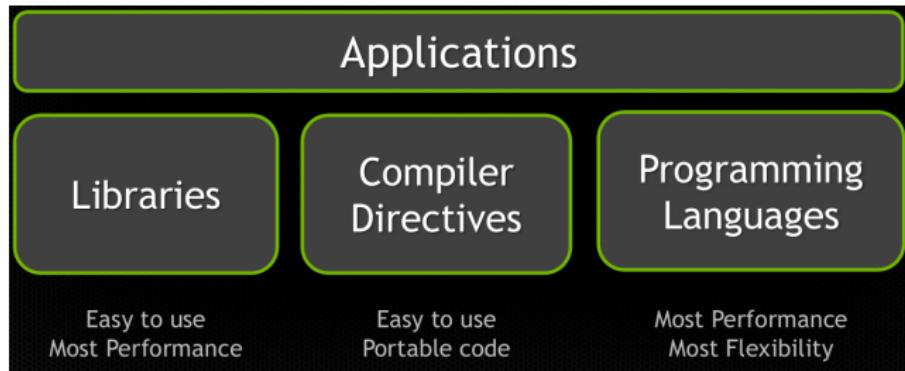
- The HOST (CPU) copies data to the DEVICE (GPU) memory
- The HOST triggers the DEVICE execution
- The DEVICE executes instructions asynchronously
- The HOST retrieves the results from the DEVICE memory



# Heterogenous computing



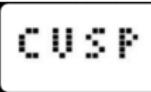
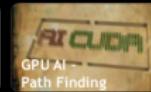
# 3 ways to accelerate GPU computing



## GPU Programming languages

- Numerical analytics: MATLAB, Mathematica, LabVIEW
- Fortran: CUDA Fortran
- C: CUDA C, OpenCL
- C++: CUDA C++, OpenCL
- Python: PyCUDA, Copperhead
- F#: Alea.cuBase
- Haskell: accelerate

# GPU libraries

<b>Linear Algebra</b> FFT, BLAS, SPARSE, Matrix	 	 
<b>Numerical &amp; Math</b> RAND, Statistics	 	
<b>Data Struct. &amp; AI</b> Sort, Scan, Zero Sum	  	
<b>Visual Processing</b> Image & Video	 