Libraries

Don't reinvent the wheel. Specialized math libraries are likely faster.

BLAS: Basic Linear Algebra Subprograms

LAPACK: Linear Algebra Package (uses BLAS)
http://www.netlib.org/lapack/ to download
http://www.netlib.org/lapack/lug/ User's GUIDE
(mac developer: in Accelerate framework)

Computational Routines

- Linear Equations
- Orthogonal Factorizations and Linear Least Squares Problems
- Generalized Orthogonal Factorizations and Linear Least Squares Problems
- Symmetric Eigenproblems
- Nonsymmetric Eigenproblems
- Singular Value Decomposition
- Generalized Symmetric Definite Eigenproblems
- Generalized Nonsymmetric Eigenproblems
- Generalized (or Quotient) Singular Value Decomposition

Intel Math Kernel Library ($ student edition)

Tridiagonal Problem

$$\begin{pmatrix} 2 & -1 & 0 & 0 & 0 & \ldots.. & 0 \\ -1 & 2 & -1 & 0 & 0 & \ldots.. & 0 \\ 0 & -1 & 2 & -1 & 0 & \ldots.. & 0 \\ 0 & 0 & -1 & 2 & -1 & \ldots. & 0 \\ . & & & & & & \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ . \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ . \\ 1 \end{pmatrix}$$

Tridiagonal Solver (on Mac)

```
#include <Accelerate/Accelerate.h>    //Mac framework
#include <stdio.h>
#define NMAX 10
using namespace std;

int main()
{
  int i, n=NMAX, one=1, info;
  double dl[NMAX-1],d[NMAX],du[NMAX- 1],b[NMAX];

/*
   Initializations
*/
for (i=0;i<n-1;i++) {
  dl[i] =-1.;
  d[i]  = 2.;
  du[i] =-1.;
  b[i]  = 1.; }

d[n-1] =2.;
b[n-1] =1.;
```

```
/* DGTSV (LAPACK) call
   all parameters are passed trough  addresses in fortran
   DGTSV  solves the equation
    A*X = B,
 where A is an n by n tridiagonal matrix, by Gaussian elimination with
 partial pivoting.
*/

dgtsv_(&n,&one,dl,d,du,b,&n,&info);      //lapack routine;
//                      Print out results
if (info==0)
    {
        for (i=0;i<n;i++) {
        printf(" i %d b(i) %le error %14.6le \n",
         i,b[i],b[i]-(i+1)*(n+1-i-1)/2.);
          }
     }
 }
```

Tridiagonal Solution

```
#include <stdio.h>
#define NMAX 10
using namespace std;

extern  "C" {                                  //C++ code to have C linkage for other C code
  void dgtsv_(int *, int *, double *, double *, double *, double *,  int *, int *);
}
extern void dgtsv_(int*, int*, double*,  double*, double*, double*, int*, int*);

int main()
{
  int i, n=NMAX, one=1, info;
  double dl[NMAX-1],d[NMAX],du[NMAX- 1],b[NMAX];
// Initializations
for (i=0;i<n-1;i++) {
  dl[i] =-1.;
  d[i]  = 2.;
  du[i] =-1.;
  b[i]  = 1.; }

d[n-1] =2.;
b[n-1] =1.;
```

lapackExample.cpp

```
// call the lapack function

dgtsv_(&n,&one,dl,d,du,b,&n,&info);

/*
   Results
*/

if (info==0)
   {
      for (i=0;i<n;i++) {
      printf(" i %d b(i) %le error %14.6le \n",
       i,  b[i],    b[i]-(i+1)*(n+1-i-1)/2.);
        }
     }
}
```

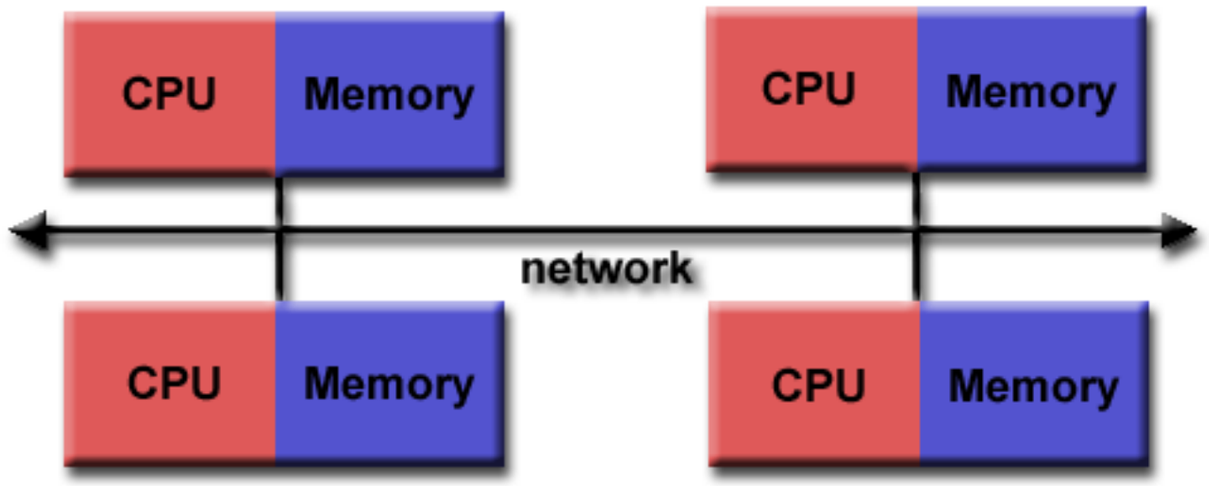
g++ lapackExample.cpp -llapack //lapack installed on machine

Message Passing Interface: MPI

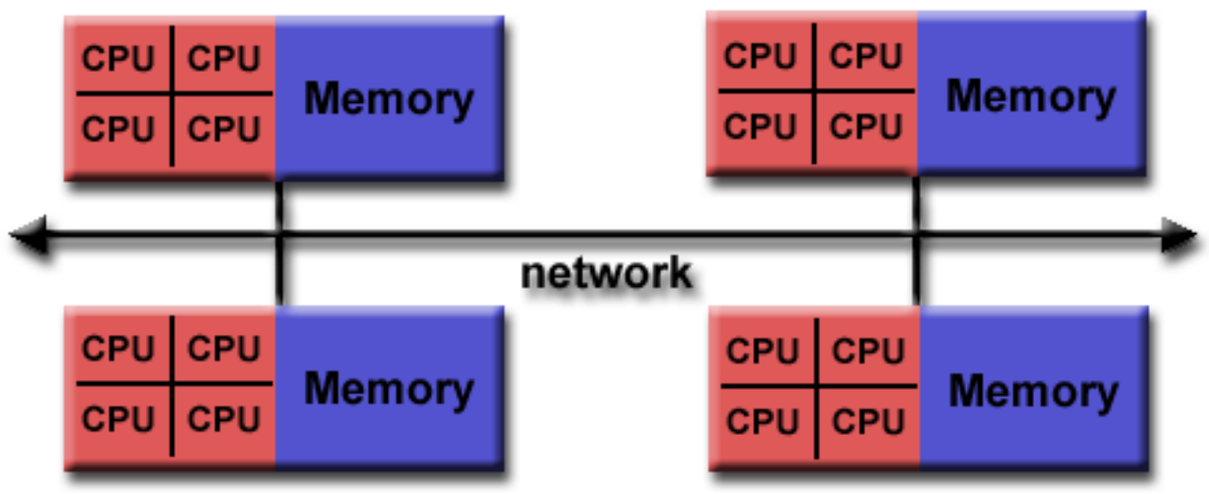
For developing parallel programs on networks.

Works for many languages: C++, FORTRAN, …

Originally developed for distributed memory machines (clusters)



Now on hybrid machines: distributed and shared memory machines



MPI

All processors have their own memory

Processes (on processors) send messages to each other

Messages have tags to direct the messages to specific processors

MPICH is a popular version of MPI, but many flavors exist

```
#include “mpi.h”          //   Hello, World
#include <stdio.h>
int main (int argc, char *argv[])
{
   MPI_Init( &argc, &argv);
   printf(“Hello, world!\n”);
   MPI_Finalize();
   return 0;
}
```

MPI sample code

http://www.open-mpi.org

mpicc is the compiler (wrapper for gcc); mpirun; mpiexec

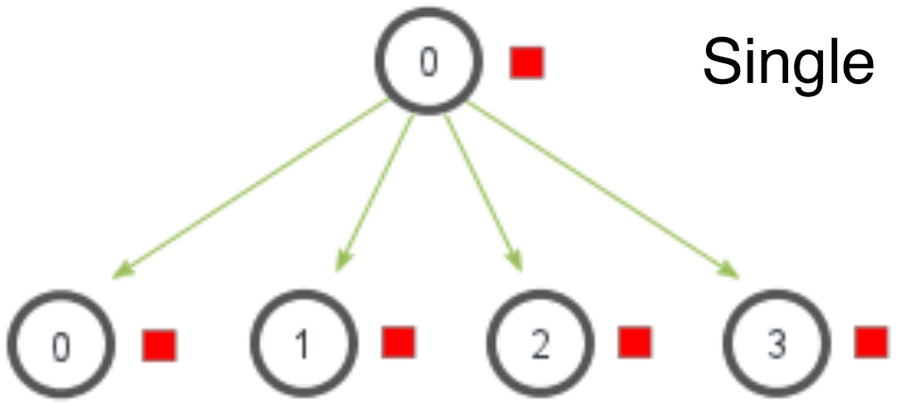
```
#include <iostream>
#include <mpi.h>
int main (int argc, char *argv())
{
  int taskid, numtasks;

  MPI_Init (&argc, &argv);   Initializes the parallel environment; args not used
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);    //number of processors
  MPI_Comm_rank(MPI_COMM_WORLD, &taskid);        //processor id
  std::cout<< “task “<< taskid<<“has started”<<endl;
  MPI_SEND(  );
  MPI_RECV(   );
  MPI_reduce(   args, MPI_COMM_WORLD);       //similar to openmp reduction
  if (taskid == MASTER)
     { stuff}
  MPI_Finalize();    //terminates the MPI session
  return 0;
}
```

MPI

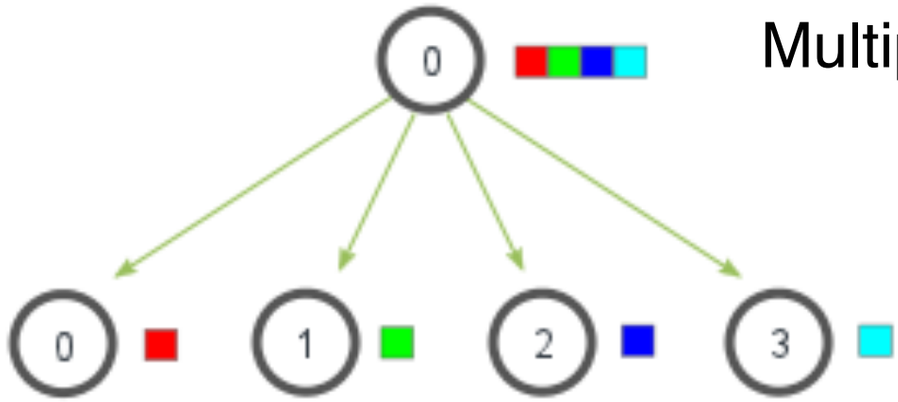
MPI_Barrier

MPI_Bcast



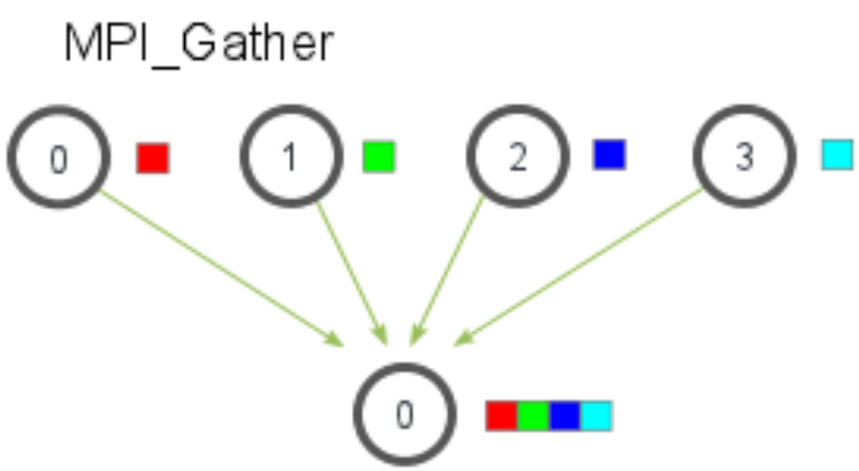
Single Instruction

MPI_Scatter



Multiple Instruction

MPI_Gather

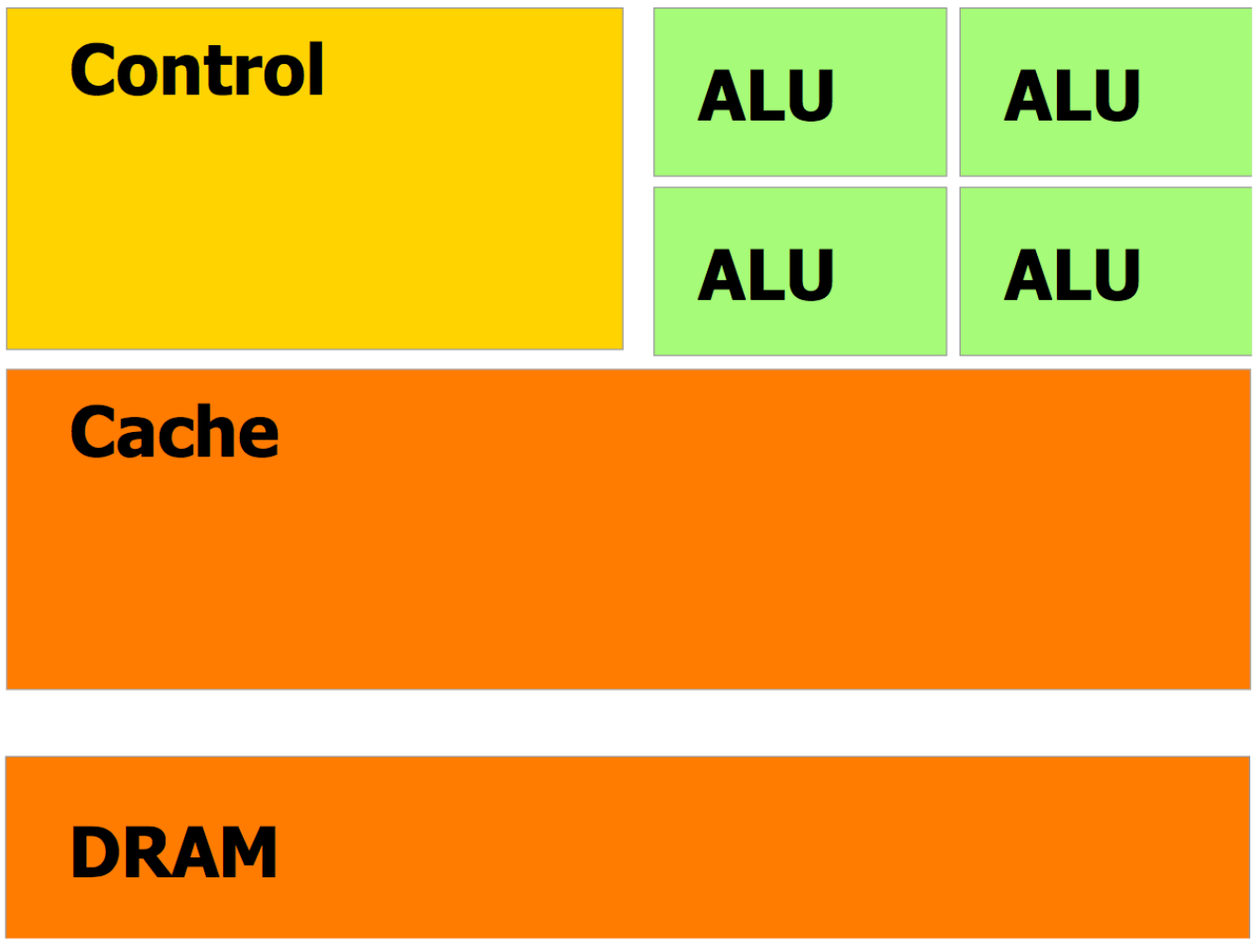


Pi

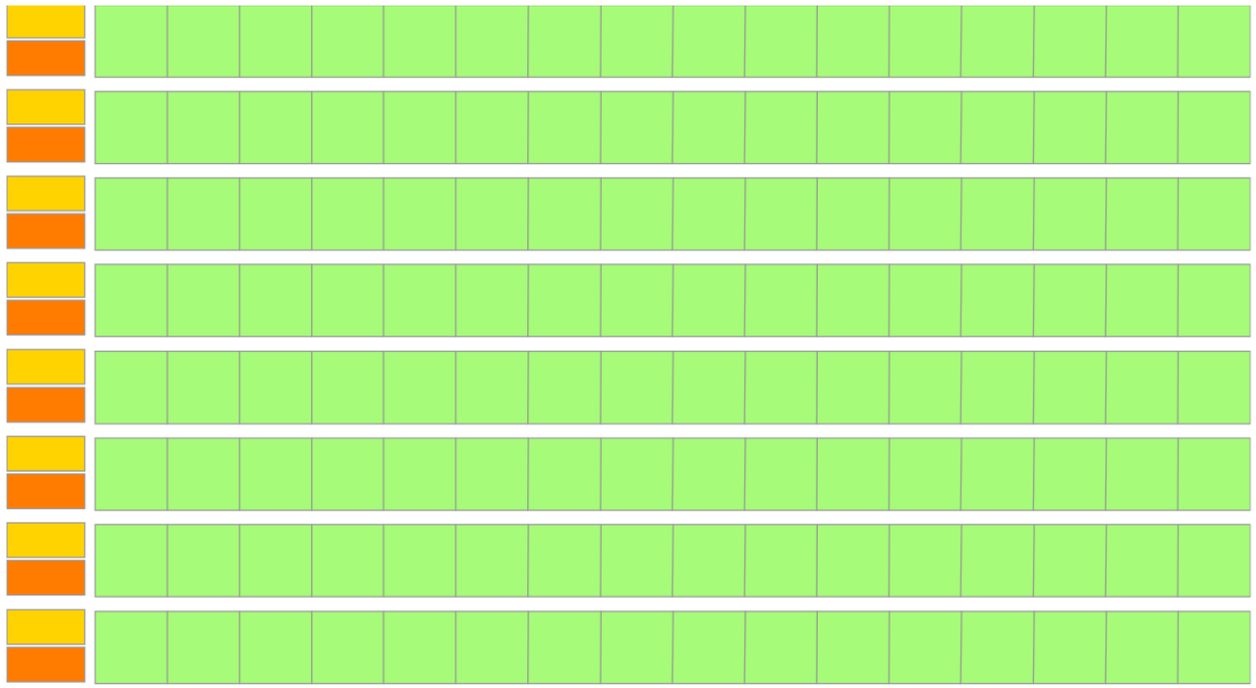
```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.14159265358979323846264643;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done)  {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```

```
h   = 1.0 / (double) n;
 sum = 0.0;
 for (i = myid + 1; i <= n; i += numprocs) {
   x = h * ((double)i - 0.5);
   sum += 4.0 / (1.0 + x*x);
 }
 mypi = h * sum;
   MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
              MPI_COMM_WORLD);
   if (myid == 0)
     printf("pi is approximately %.16f, Error is %.16f\n",
            pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```

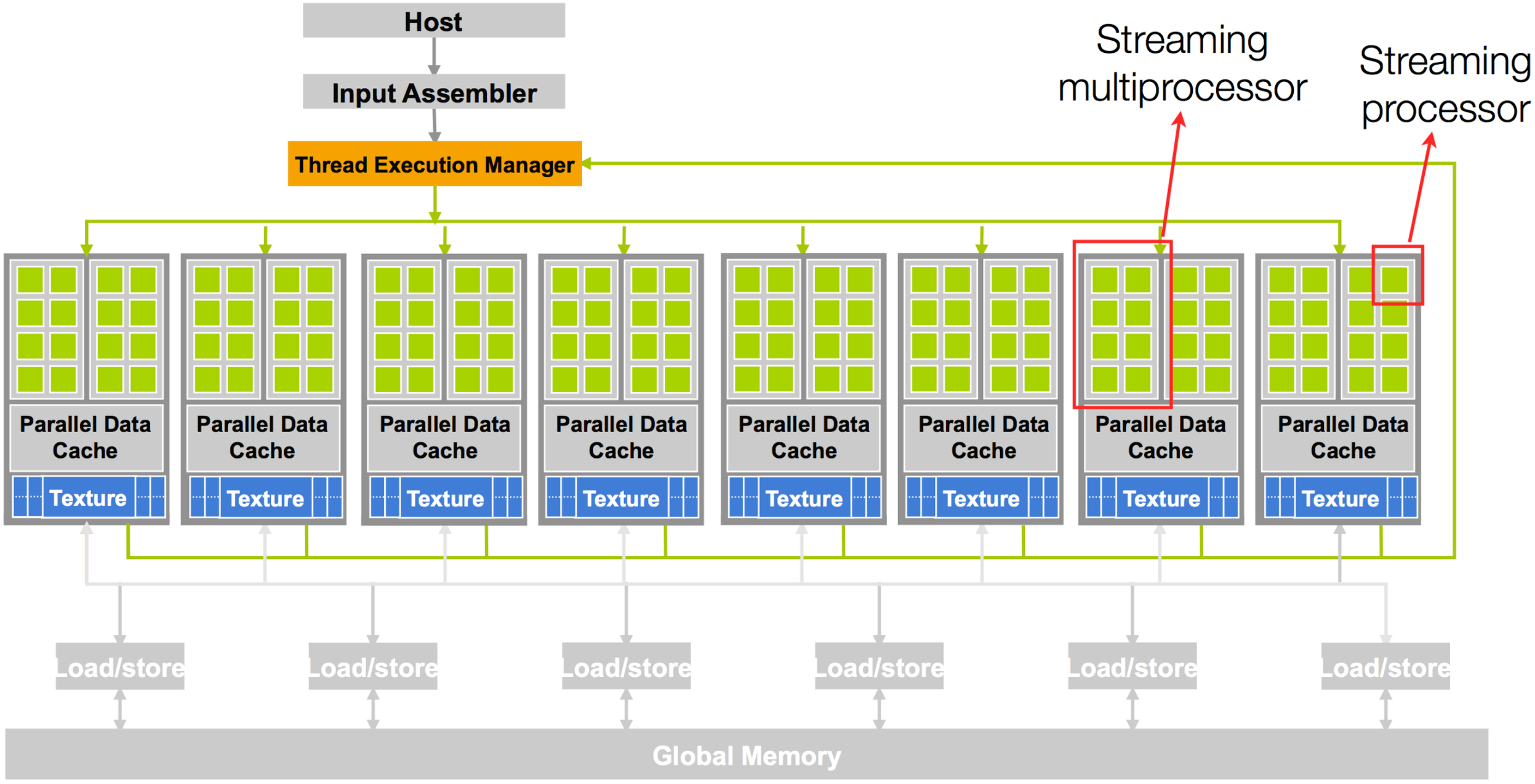
CPU vs GPU



CPU



GPU



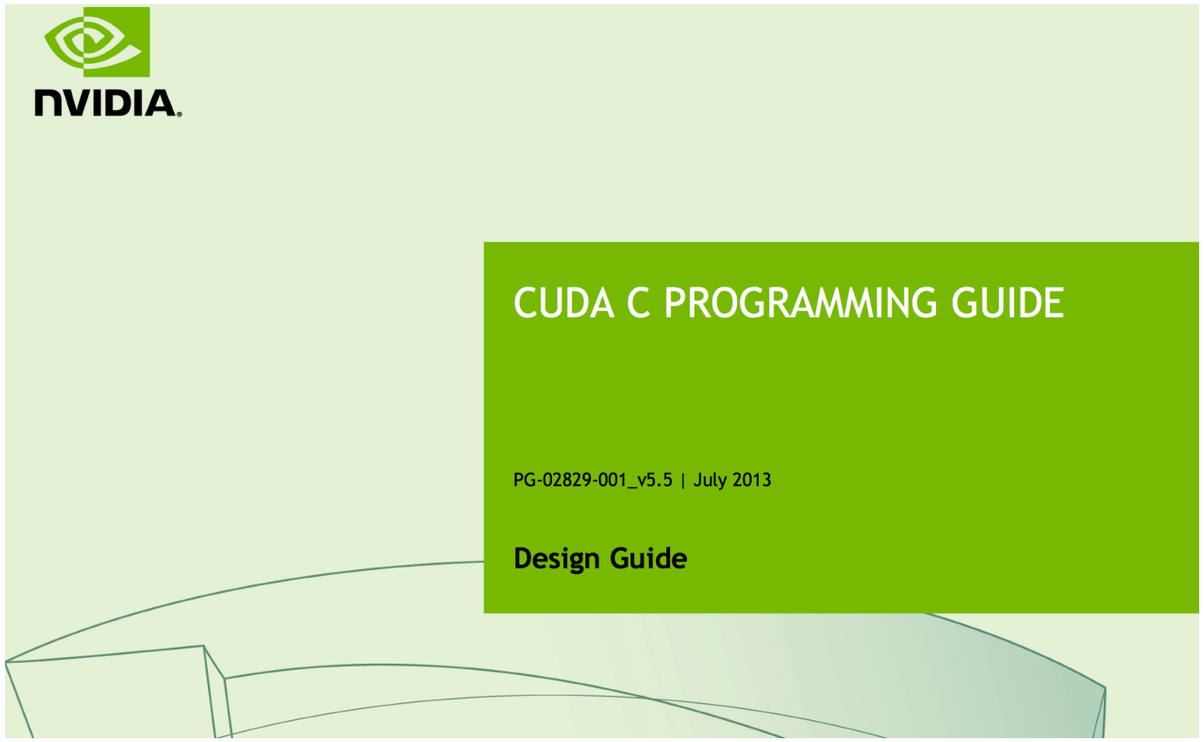
Streaming multiprocessor (SM) has 8 streaming processors (SP)

CUDA

Graphics Processing Units: important for windowed operating systems, 3D drawings (Silicon Graphics and OpenGL), video games (late 1980s and 1990s). Nvidia GeForce 256 (2001) allowed OpenGL and Direct X allowed programming the graphics card.

GPU languages:

GPGPU
CUDA with nvcc compiler (2007)
OpenCL
OpenACC like OpenMP



References:

NVIDIA
http://docs.nvidia.com/cuda/index.html

https://developer.nvidia.com/suggested-reading

CUDA SDK

Examples: /Developer/NVIDIA/CUDA-6.5/samples

0_Simple
1_Utilities
2_Graphics
3_Imaging
4_Finance
5_Simulations
6_Advanced
7_CUDALibraries

Try deviceQueryDrv.cpp to find out about your graphics card

/Developer/NVIDIA/CUDA-5.5/samples/1_Utilities/deviceQueryDrv
Cuda Capability 3.0; global memory: 1024 Mb; 384 CUDA cores

In 5_Simulations, make the fluidsGL and the Particles programs and run.

```
Device 0: "GeForce GT 650M"
  CUDA Driver Version:                           6.0
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 1024 MBytes (1073414144 bytes)
  ( 2) Multiprocessors, (192) CUDA Cores/MP:     384 CUDA Cores
  GPU Clock rate:                                900 MHz (0.90 GHz)
  Memory Clock rate:                             2508 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 262144 bytes
  Max Texture Dimension Sizes                    1D=(65536) 2D=(65536, 65536) 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z):    (2147483647, 65535, 65535)
  Texture alignment:                             512 bytes
  Maximum memory pitch:                          2147483647 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   Yes
  Alignment requirement for Surfaces:            Yes
```

Maximum number of active threads

(Max_num_threads_SM)* (Num_of_SM)

2048 *2 = 4096 at once

Multi-core CPU vs GPU

Multi-core computer

- Emphasize multiple full-blown processor cores, implementing the complete instruction set of the CPU
- The cores are out-of-order implying that they could be doing different tasks
- They may additionally support hyperthreading with two hardware threads
- Designed to maximize the execution speed of sequential programs

GPU

- Typically have hundreds or thousands of cores
- Cores are heavily multithreaded, in-order, and single-instruction issue processors
- Each core shares control and instruction cache with seven other cores

Nvidia GPU

State-of-the-art Kepler Cards

Features	Tesla K40	Tesla K20X	Tesla K20	Tesla K10
Number and Type of GPU	1 Kepler GK110B	1 Kepler GK110		2 Kepler GK104s
Peak double precision floating point performance	1.43 Tflops	1.31 Tflops	1.17 Tflops	0.19 Tflops
Peak single precision floating point performance	4.29 Tflops	3.95 Tflops	3.52 Tflops	4.58 Tflops
Memory bandwidth (ECC off)	288 GB/sec	250 GB/sec	208 GB/sec	320 GB/sec
Memory size (GDDR5)	12 GB	6 GB	5 GB	8 GB
CUDA cores	2880	2688	2496	2 x 1536

Maxwell cards just announced (9/2014).

Other cards (GeForce, Quadro, Tesla): https://developer.nvidia.com/cuda-gpus

Each card has a Compute Capability: 1 -> 3.5

C++ vectorAdd.cpp

```
#include <iostream>
#define SIZE 1024
using namespace std;

void vectorAdd(int *a, int *b, int *c, int n)
 {
   for (int i = 1; i < n; i++)
     c[i] = a[i] + b[i];
  }

int main()
 {
   int * a;
   int * b;
   int * c;
   a = new int[SIZE];
   b = new int[SIZE];
   c = new int[SIZE];
   for (int i = 0; i < SIZE; i++)     //initialize vectors
    {
      a[i]=i;
      b[i]=i;
      c[i]=0;
      }
```

vectorAdd.cpp (2)

```
vectorAdd(a,b,c, SIZE);
   for(int i = 0; i<10; i++) //print out first 10
    cout<< "c[" <<i<<"] = "<<c[i]<<endl;

    delete [] a;
    delete [] b;
    delete [] c;

}
```

Now use GPU for this problem

We have to modify the program four ways:

1. Parallelize the vectorAdd function for GPU

2. Allocate memory on GPU and move data over

3. Enable vectorAdd call to work on GPU

4. Bring results back from GPU

CUDA FUNCTION

```
/* C++ VECTOR ADD:  c = a + b
*/
#include <iostream>
#define SIZE 1024
using namespace std;

__global__ void vectorAdd(int *a, int *b, int *c, int n)
 {
    int i=threadIdx.x;  //replaces for loop as whole function is parallel
    if (i<n)            //only do as many threads as you have data
       c[i] = a[i] + b[i];
}
```

```
int main()
 {
   int * a;
   int * b;
   int * c;
   int *d_a; //device versions of a, b, c
   int *d_b;
   int *d_c;
   a = new int[SIZE];
   b = new int[SIZE];
   c = new int[SIZE];
cudaMalloc( &d_a, SIZE*sizeof(int));  //address_of operator
cudaMalloc( &d_b, SIZE*sizeof(int));
cudaMalloc( &d_c, SIZE*sizeof(int));
for (int i = 0; i < SIZE; i++)    //initialize vectors
 {
   a[i]=i;
   b[i]=i;
   c[i]=0;
   }
 cudaMemcpy(d_a, a, SIZE*sizeof(int), cudaMemcpyHostToDevice);
 cudaMemcpy(d_b, b, SIZE*sizeof(int), cudaMemcpyHostToDevice);
 cudaMemcpy(d_c, c, SIZE*sizeof(int), cudaMemcpyHostToDevice);
```

```
vectorAdd<<<1,SIZE>>>(d_a,d_b,d_c, SIZE);   //CUDA kernel

cudaMemcpy(c, d_c, SIZE*sizeof(int), cudaMemcpyDeviceToHost);
for(int i = 0; i<10; i++) //print out first 10
 cout<< "c[" <<i<<"] = "<<c[i]<<endl;

 delete [] a;
 delete [] b;
 delete [] c;
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

}
```

---

vectorAdd<<<1,SIZE>>>(d_a,d_b,d_c, SIZE);   //CUDA kernel

<<< num_blocks, threads_per_block >>>
Here 1 block, 1024 threads in block.