

Week 4

Part I Vector supercomputing

Computer architectures

- ◆ CPU
 - ◆ CISC
 - ◆ RISC
 - ◆ Vector CPUs

- ◆ Memory
 - ◆ Main memory
 - ◆ Cache

- ◆ Optimization for vector CPUs

CISC

- ◆ "Complex instruction Set Computer"
 - ◆ Example: Pentium

- ◆ Faster performance obtained by providing more complex and high level instructions on the CPU

- ◆ Disadvantage:
 - ◆ large complex CPUs
 - ◆ large power consumption

RISC and superscalar CPUs

- ◆ "Reduced Instruction Set Computer"
 - ◆ Examples: Power, PowerPC, Alpha, Mips, Sparc

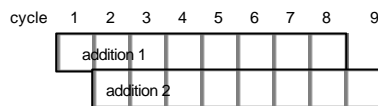
- ◆ Uses lower levels instructions, but
 - ◆ Instructions executed faster
 - ◆ Several instructions executed concurrently
 - ◆ Instructions can be pipelined

- ◆ Disadvantages
 - ◆ Optimization for speed is responsibility of compiler

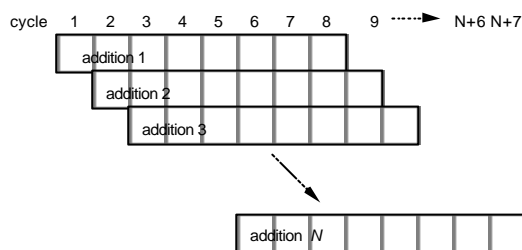
- ◆ Advantages
 - ◆ Faster, smaller, needs less power

Pipelining

- ◆ One addition takes eight cycles
- ◆ But second addition can start one cycle after first



- ◆ N additions take only $N+7$ cycles



Memory bottlenecks

- ◆ One pipeline can process one floating point operation per cycle
- ◆ Often more than one pipeline per CPU
 - ◆ Pentium: 1 single precision operation per cycle
 - ◆ Alpha EV6: 2 double precision operation per cycle
 - ◆ PowerPC G4: 8 single precision operations per cycle
- ◆ Problem
 - ◆ `for (int j=0; j<N; ++j) a[j]=b[j]+k[j]`
 - ◆ 500 MHz PowerPC G4 can do 4 additions per cycle
 - ◆ Needs to load 16 GByte per second!
 - ◆ Needs to store 8 Gbyte per second!
- ◆ PC or workstation memory cannot do that!

Memory hierarchies: example Alpha 21161

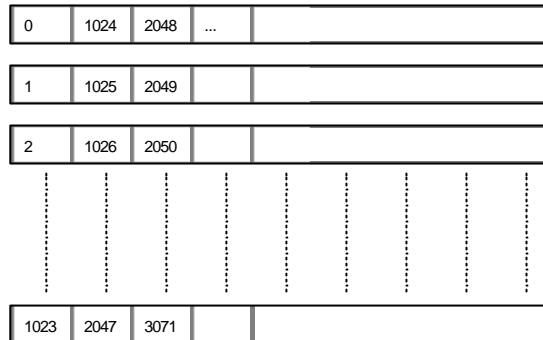
- ◆ Fast memory is very expensive
- ◆ Solution: add a hierarchy of caches
- ◆ This helps if a program works most of the time only on a small amount of data
- ◆ Need to write programs to make use of data-locality
- ◆ Can never reach peak performance!!!
- ◆ Main memory
 - ◆ Access time 220 ns
- ◆ L3 cache, 8 MByte
 - ◆ Access time 30 ns
- ◆ L2 cache 96 KByte
 - ◆ Access time 5 ns
- ◆ L1 cache 16 Kbyte
 - ◆ Access time 4 ns
- ◆ Registers, several hundred bytes
 - ◆ Access time 2 ns

Solution: Vector CPUs

- ◆ In addition to pipelined processing units we need
 - ◆ vector registers
 - ◆ pipelined memory access
- ◆ Vector registers
 - ◆ 64, 128 or 256 words long (64 bits per word)
 - ◆ Provide full bandwidth from registers to processing units
- ◆ Pipelined memory access
 - ◆ Provides reading of one or two words (64 bits per word) per cycle
 - ◆ Provides writing of one or two words (64 bits per word) per cycle
 - ◆ Memory banks used to mask slow access times

Memory banks of vector computers

- ◆ Memory striped over 1024 memory banks
- ◆ Simultaneous access to all 1024 banks possible
- ◆ Can read and write one number per cycle
- ◆ Can reach peak performance in pipelined loops



Vectorizing a program

- ◆ Vector computers can "vectorize" long loops efficiently
 - ◆ `double a[N], b[N], c[N];`
 - `for (int i=0;i<N;i++)`
 - `a[i]=b[i]+c[i];`
 - ◆ Needs one cycle per number for large N
- ◆ Vectorization can be extended to other cases
- ◆ We need to learn about
 - ◆ Loop interchange
 - ◆ Loop fusion/splitting
 - ◆ Loop splitting
 - ◆ Gather/scatter
 - ◆ Dependencies
 - ◆ Bank conflicts

Loop interchange

- ◆ The two loops

- ◆

```
double a[1000][10], b[1000][10], c[1000][10];
for (int i=0;i<1000;i++)
  for (int j=0;j<10;j++)
    a[i][j]=b[i][j]+c[i][j];
```

- ◆ Are better rearranged to

- ◆

```
double a[10][1000],b[10][1000],c[10][1000];
for (int i=0;i<10;i++)
  for (int j=0;j<1000;j++)
    a[i][j]=b[i][j]+c[i][j];
```

- ◆ The inner loop, which can be vectorized, gets longer!

- ◆ Smart compilers can do that automatically, but don't count on it

Loop fusion

- ◆ The two loops

- ◆

```
double a[10][1000],b[10][1000],c[10][1000];
for (int i=0;i<10;i++)
  for (int j=0;j<1000;j++)
    a[i][j]=b[i][j]+c[i][j];
```

- ◆ Are better fused into one

- ◆

```
double a[10000],b[10000],c[10000];
for (int i=0;i<10000;i++)
  a[i] =b[i] +c[i];
```

- ◆ Again the loop gets longer!

- ◆ Smart compilers can do that automatically, but don't count on it

Loop splitting

- ◆ Calculate the norm of 3-vectors

```
double a[1000][3], norm[1000];
for (int i=0;i<1000;i++) {
    norm[i]=0.;
    for (int j=0;j<3;j++)
        norm[i]+=a[i][j]*a[i][j];
    norm[i]=sqrt(norm[i]);
}
```

- ◆ Is better split into three loops

```
double a[1000][3], norm[1000];
for (int i=0;i<1000;i++)
    norm[i]=0;
for (int j=0;j<3;j++)
    for (int i=0;i<1000;i++)
        norm[i]+=a[i][j]*a[i][j];
for (int i=0;i<1000;i++)
    norm[i]=sqrt(norm[i]);
```

Loop unrolling

- ◆ Calculate the norm of 3-vectors

```
double a[1000][3], norm[1000];
for (int i=0;i<1000;i++) {
    norm[i]=0.;
    for (int j=0;j<3;j++)
        norm[i]+=a[i][j]*a[i][j];
    norm[i]=sqrt(norm[i]);
}
```

- ◆ The inner loop can be unrolled

```
double a[1000][3], norm[1000];
for (int i=0;i<1000;i++)
    norm[i]=a[i][1]*a[i][1]+a[i][2]*a[i][2]+a[i][3]*a[i][3];
```

- ◆ Smart compilers can do that automatically, but don't count on it

Bank conflicts

◆ The code

```

◆ const int N=1024;
double a[N][N], b[N][N], c[N][N];
for (int i=0;i<N;i++)
  for (int j=0;j<N;j++) {
    a[i][j]=0.;
    for (int k=0;k<N;k++)
      a[i][j] += b[i][k]*c[k][j];
  }

```

◆ Is as slow as on a workstation

- ◆ **Reason:** in the inner loop the stride for `c` is 1024, which is the number of banks. The same bank is used all the time, thus no pipelining possible
- ◆ **Solution:** set `N=1025`

◆ Take care:

- ◆ On workstations/PCs **prefer** strides of 2^n
- ◆ On vector computers **avoid** strides of 2^n

Dependencies

◆ The linear congruential random number generator

```

◆ long rnd[N]
  rnd[0]=667790;
  long a=16807;
  long m=(1<<32)-1; // 2^32-1
  for (int i=1;i<N;i++)
    rnd[i]=(rnd[i-1]*a)%m;

```

- ◆ Cannot be vectorized
- ◆ Each iteration has to wait for the previous one to finish

◆ Lagged Fibonacci generators

```

◆ for (int i=p+1;i<N;i++) {
  double t=rnd[i-p]+rnd[i-q];
  rnd[i]=t-double(int(t));
}

```

- ◆ Can be vectorized when `p` and `q` are larger than the vector length

Gather operations

- ◆ In the loop
 - ◆ `for (int i=0;i<N;i++)`
 `y[i] = x[index[i]]; // gather operation`
 - ◆ The memory access patterns for x are not known at compile time
- ◆ There can be bank conflicts
 - ◆ E.g. `index[0]` and `index[1]` point to the same bank
 - ◆ This slows the pipeline
- ◆ Solution: gather hardware
 - ◆ If a bank conflict would occur, look ahead and reorder the iterations to avoid as many bank conflicts as possible
 - ◆ Needs to be done by the hardware at run time
 - ◆ Very expensive
 - ◆ Available only on on high-end vector machines

Scatter operations

- ◆ In the loop
 - ◆ `for (int i=0;i<N;i++)`
 `x[index[i]]=y[i]; // scatter operation`
- ◆ Bank conflicts solved by scatter hardware
- ◆ Dependencies are a problem
 - ◆ Cannot be vectorized as `index[i]` could have the same value twice!
- ◆ If the programmer knows that there are no dependencies, he can help the compiler:
 - ◆ `#pragma _CRI ivdep`
 `for (int i=0;i<N;i++)`
 `x[index[i]]=y[i]; // scatter operation`
 - ◆ Can be vectorized
 - ◆ However incorrect results if there are dependencies, be careful!

Non-vectorizable functions

◆ Non-vectorizable function calls

```
◆ double x[N];
  for (int i=0;i<N;i++) {
    x[i] = fabs(x[i]);
    cout << i << " " << x[i] << "\n";
  } // does not vectorize
```

◆ Does not vectorize, because of the call to the output function

◆ Vectorizable function calls

```
◆ double x[N];
  for (int i=0;i<N;i++) {
    x[i] = fabs(x[i]);
  } // vectorizes
```

◆ vectorizes, because there is a vectorizing `fabs` function on the Cray

Branches in vector loops

◆ If-statements can be vectorized

```
◆ double x[N];
  for (int i=0;i<N;i++)
    if (x[i]>0.) {
      // do work on x[i]
    }
    else {
      // also do work on x[i]
    }
```

◆ Can be vectorized

◆ Performance penalty on vector CPUs (not on scalar ones) as both parts are executed for all N entries, although some indices are masked out off the vector

◆ Faster method if a lot of work needs to be done in the loop:

```
◆ Sort indices
◆ double x[N];
  int positive[N], negative[N];
  int pos_n=0, neg_n=0;

  // sort the indices, does not vectorize
  for (int i=0;i<N;i++)
    if (x[i]>0.)
      positive[pos_n++]=i;
    else
      negative[neg_n++]=i;

  for (int i=0;i<pos_n;i++) {
    // do a lot of work on x[positive[i]]
  }
  for (int i=0;i<neg_n;i++) {
    // do a lot of work on x[negative[i]];
  }
```

Other comments

- ◆ Vectorization easier for simple loops
 - ◆ Try to split a loop into a few loops if it does not vectorize

- ◆ Do not use integer arithmetic on vector machines
 - ◆ Adding integers is 10 times more expensive than adding real numbers
 - ◆ 64-bit integer arithmetic is performed using 128-bit floating point units

- ◆ Look at the compiler listings to see
 - ◆ which loops vectorize
 - ◆ Which loops do not vectorize, and why not
 - ◆ Which function calls are inlined

- ◆ Most comments also valid for superscalar RISC machines

Exercise

- ◆ Change the heat equation solver program to 2D

- ◆ Port it to the Cray

- ◆ Vectorize the code

- ◆ Compare the execution times of scalar and vector code

Cray SV1
at ETHZ

