PITTSBURGH SUPERCOMPUTING CENTER

Manual and Compiler Optimizations

Shawn T. Brown, PhD.

Director of Public Health Applications

Pittsburgh Supercomputing Center, Carnegie Mellon University

Introduction to Performance

Optimization

- Real processors have
  - registers, cache, parallelism, ... they are bloody complicated
- Why is this your problem?
  - In theory, compilers understand all of this and can optimize your code; in practice they don't.
  - Generally optimizing algorithms across all computational architectures is an impossible task, hand optimization will always be needed.
- We need to learn how...
  - to measure performance of codes on modern architectures
  - to tune performance of the codes by hand (32/64 bit commodity processors) and use compilers
  - to understand parallel performance

Performance

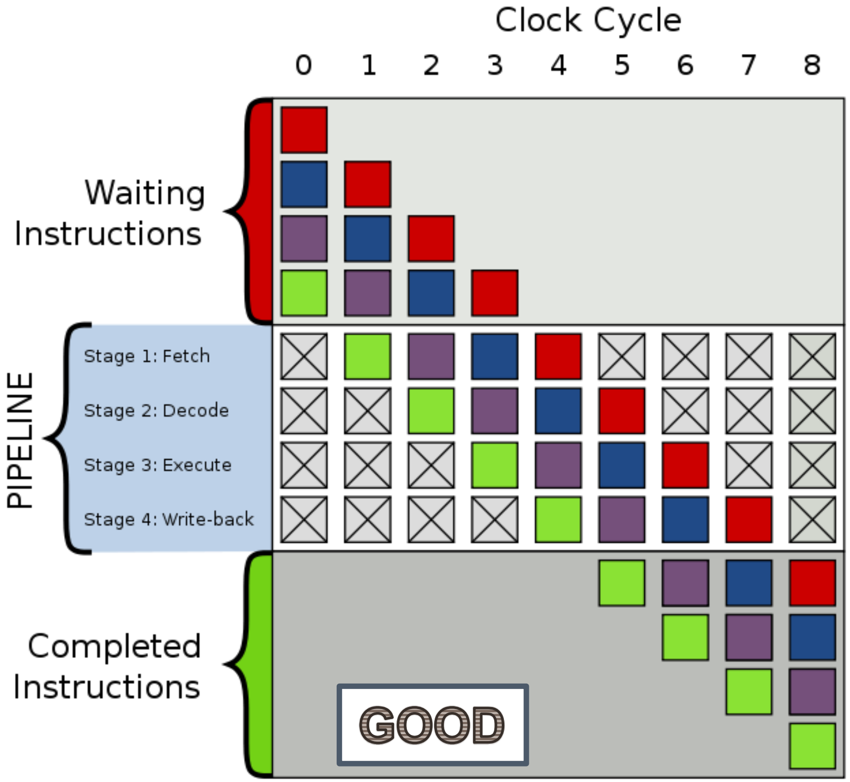
- The peak performance of a chip
  - The number of theoretical floating point operations per second
    - e.g. 2.8 Ghz Core-i7 has 4 cores and each core can do theoretically 4 flops per cycle, for a peak performance of 44.8 Gflops
- Real performance
  - Algorithm dependent, the actually number of floating point operations per second
    - Generally, most programs get about 10% or lower of peak performance
    - 40% of peak, and you can go on holiday
- Parallel performance
  - The scaling of an algorithm relative to its speed on 1 processor.

Serial Performance

- On a single processor (core), how fast does the algorithm complete.

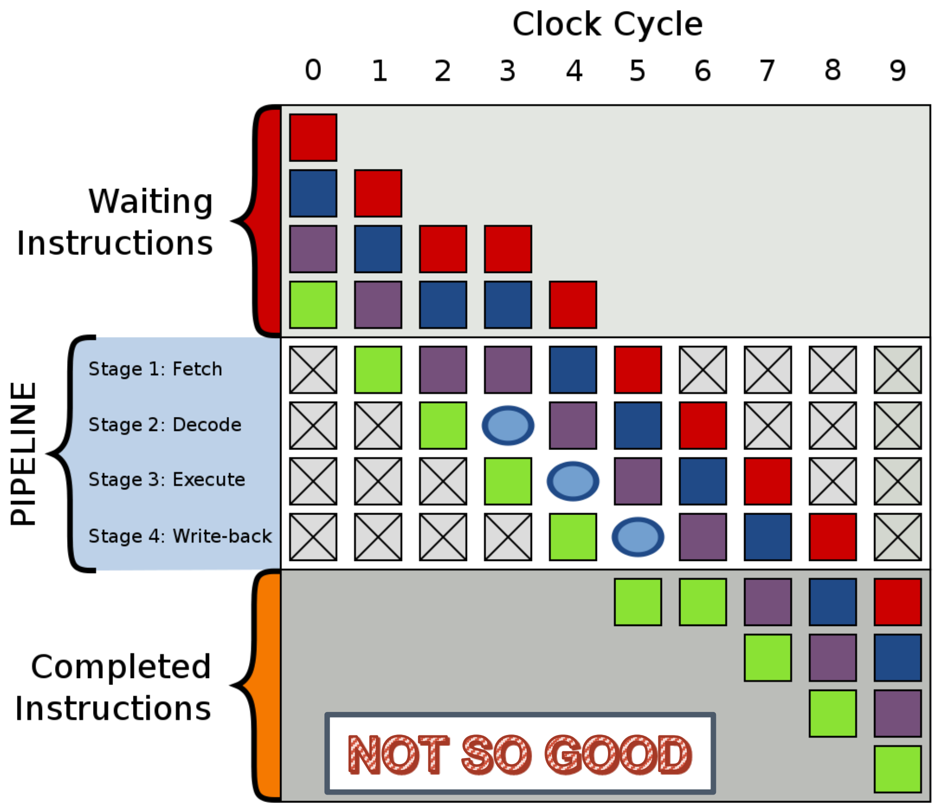
- Factors:
  - Memory
  - Processing Power
  - Memory Transport
  - Local I/O
  - Load of the Machine
  - Quality of the algorithm
  - Programming Language

Pipelining



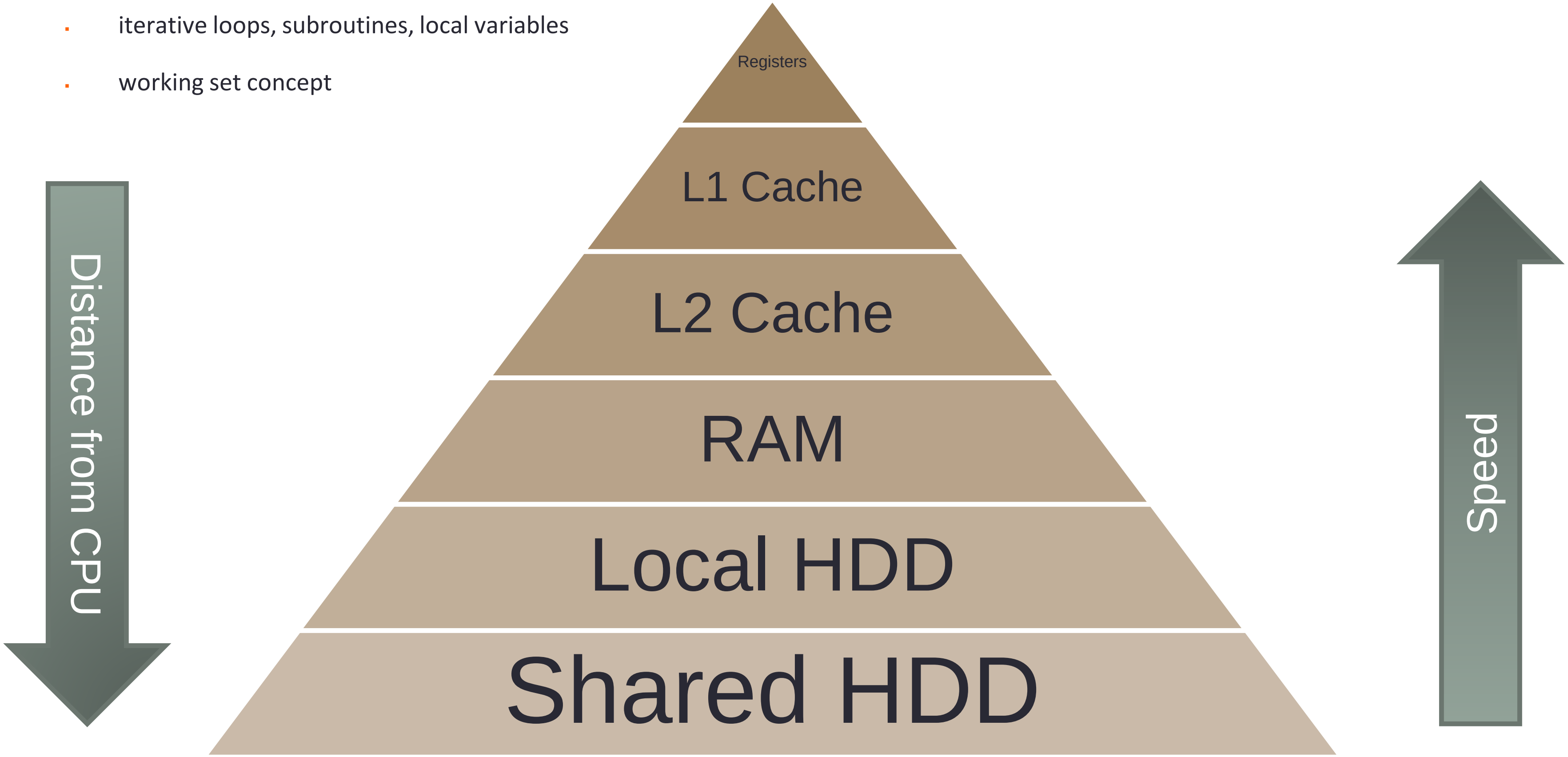
- Pipelining allows for a smooth progression of instructions and data to flow through the processor
- Any optimization that facilitate pipelining will speed the serial performance of your code.
- As chips support more SSE like character, filling the pipeline is more difficult.

- Stalling the pipeline slows codes down
  - Out of cache reads and writes
  - Conditional statements

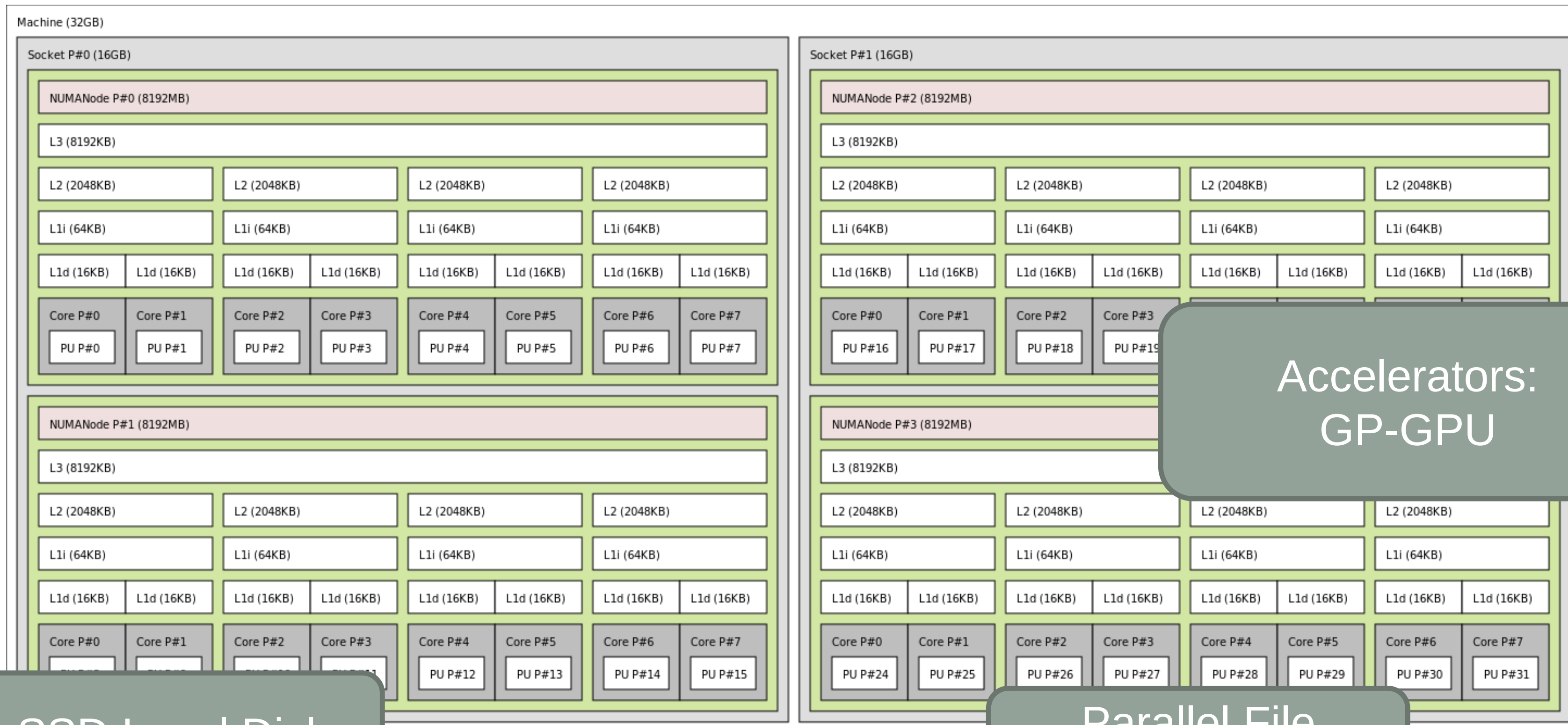


Memory Locality

- **Effective use of the memory heirarchy can facilitate good pipelining**
- **Temporal locality:**
  - Recently referenced items (instr or data) are likely to be referenced again in the near future
  - iterative loops, subroutines, local variables
  - working set concept
- **Spatial locality:**
  - programs access data which is near to each other:
  - operations on tables/arrays
  - cache line size is determined by spatial locality



Welcome to the complication....



Accelerators: GP-GPU

SSD Local Disk

Parallel File Systems

Understanding the Hardware

Variety is the spice of life…

Intel® Xeon Phi™ Coprocessor Family Reference Table

SKU #	Form Factor, Thermal	Peak Double Precision	Max # of Cores	Clock Speed (GHz)	GDDR5 Memory Speeds (GT/s)	Peak Memory BW	Memory Capacity (GB)	Total Cache (MB)	Board TDP (Watts)	Process
SE10P (special edition)	PCIe Card, Passively Cooled	1073 GF	61	1.1	5.5	352	8	30.5	300	22nm
SE10X (special edition)	PCIe Card, No Thermal Solution	1073 GF	61	1.1	5.5	352	8	30.5	300	
5110P	PCIe Card, Passively Cooled	1011 GF	60	1.053	5.0	320	8	30	225	
	Card, Actively Cooled	>1 TF	Disclosed at 3100 series launch (H1'13)		5.0	240	6	28.5	300	
	Cle Card, ively Cooled	> 1 TF			5.0	240	6	28.5	300	

PCIe Card, Actively Cooled

PCIe Card, Passively Cooled

information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information. Contact our distributor to obtain the latest specification before placing your product order.
names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third tel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user. All ates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

Intel® Core i7-800 Desktop Processor Series

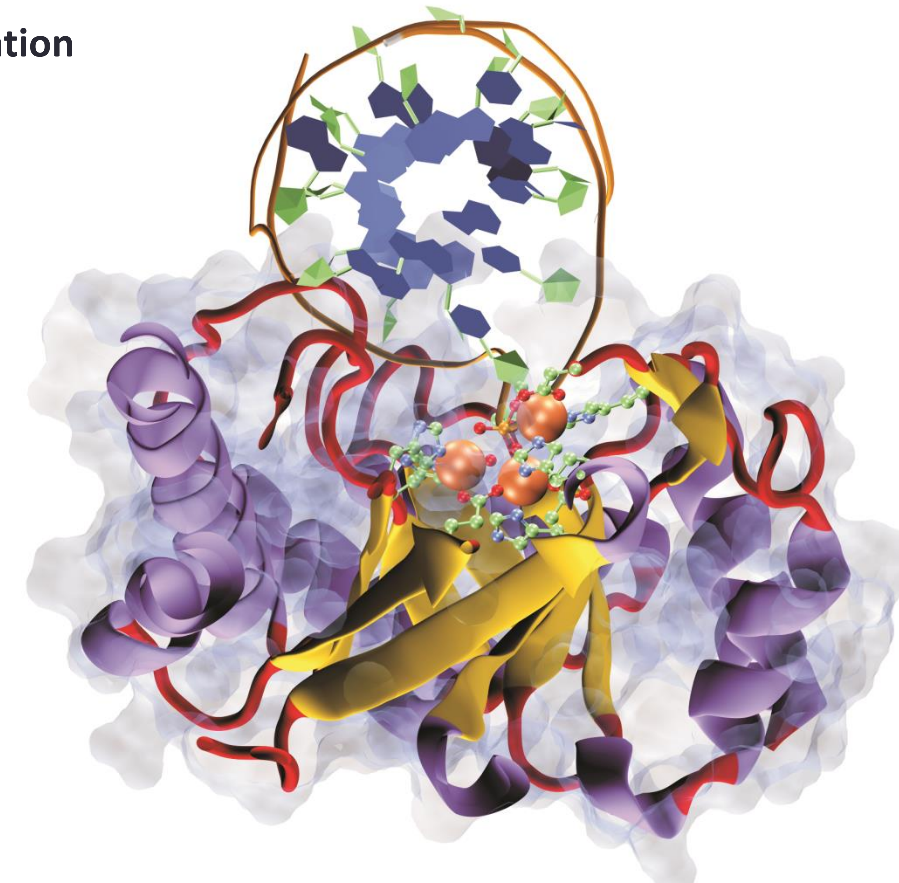
Processor Number	Frequency Type	Clock GHz	CTP	GFLOP	APP 1-way	APP 2-way	APP 4-way
i7-860	Base	2.8	84933	44.8	0.01344	0.02688	0.05376
	Single Core Max Turbo	3.46	105135	55	0.0166368	0.0332736	0.0665472
	GPU ONLY	N/A	N/A	N/A	N/A	N/A	N/A
i7-860S	Base	2.53	76834	40.528	0.0121584	0.0243168	0.0486336
	Single Core Max Turbo	3.46	105135	55	0.0166368	0.0332736	0.0665472
	GPU ONLY	N/A	N/A	N/A	N/A	N/A	N/A
i7-870	Base	2.93	88968	46.928	0.0140784	0.0281568	0.0563136
	Single Core Max Turbo	3.6	109200	58	0.01728	0.03456	0.06912
	GPU ONLY	N/A	N/A	N/A	N/A	N/A	N/A
i7-870S	Base	2.66	80869	42.656	0.0127968	0.0255936	0.0511872
	Single Core Max Turbo	3.6	109200	58	0.01728	0.03456	0.06912
	GPU ONLY	N/A	N/A	N/A	N/A	N/A	N/A
i7-875K	Base	2.93	88968	46.928	0.0140784	0.0281568	0.0563136
	Single Core Max Turbo	3.6	109200	58	0.01728	0.03456	0.06912
	GPU ONLY	N/A	N/A	N/A	N/A	N/A	N/A
i7-880	Base	3.06	93002	49.056	0.0147168	0.0294336	0.0588672
	Single Core Max Turbo	3.73	113234	60	0.0179184	0.0358368	0.0716736
	GPU ONLY	N/A	N/A	N/A	N/A	N/A	N/A

TECHNICAL SPECIFICATIONS	TESLA K10[a]	TESLA K20	TESLA K20X
Peak double precision floating point performance (board)	0.19 teraflops	1.17 teraflops	1.31 teraflops
Peak single precision floating point performance (board)	4.58 teraflops	3.52 teraflops	3.95 teraflops
Number of GPUs	2 x GK104s	1 x GK110	
Number of CUDA cores	2 x 1536	2496	2688
Memory size per board (GDDR5)	8 GB	5 GB	6 GB
Memory bandwidth for board (ECC off)[b]	320 GBytes/sec	208 GBytes/sec	250 GBytes/sec
GPU computing applications	Seismic, image, signal processing, video analytics	CFD, CAE, financial computing, computational chemistry and physics, data analytics, satellite imaging, weather modeling	
Architecture features	SMX	SMX, Dynamic Parallelism, Hyper-Q	
System	Servers only	Servers and Workstations	Servers only

Fitting algorithms to hardware…and vice versa

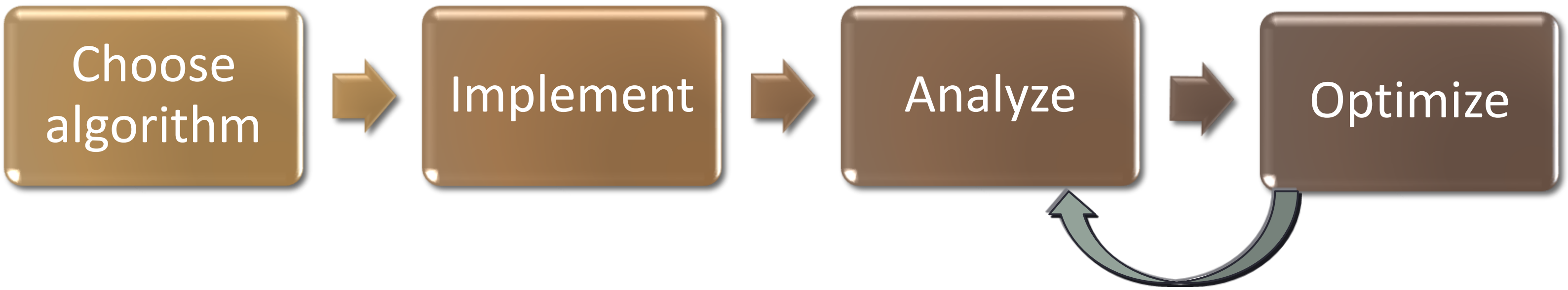
Molecular dynamics simulations on Application Specific Integrated Circuit (ASIC)

DE Shaw Research



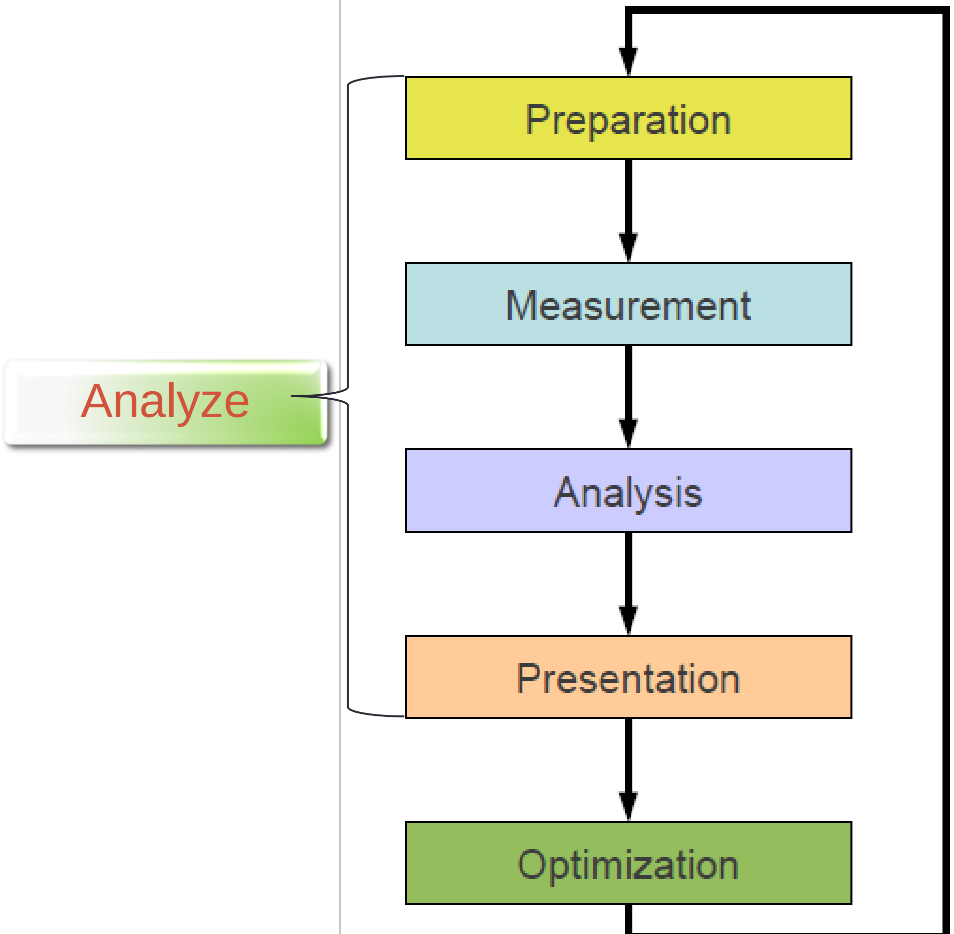
Ivaylo Ivanov, Andrew McCammon, UCSD

Code Development and Optimization Process



- Choice of algorithm most important consideration (serial and parallel)
- Highly scalable codes must be designed to be scalable from the beginning!
- Analysis may reveal need for new algorithm or completely different implementation rather than optimization
- Focus of this lecture: performance and using tools to assess parallel performance

Performance



- Prepare application, insert extra code (probes/hooks)
- Collection of data relevant to performance analysis
- Calculation of metrics, identification of performance metrics
- Visualization of results in an intuitive/understandable form
- Elimination of performance problems

Christian Rössel, Jüelich

Philosophy...

- When you are charged with optimizing an application...
  - Don't optimize the whole code
    - Profile the code, find the bottlenecks
    - They may not always be where you thought they were
  - Break the problem down
    - Try to run the shortest possible test you can to get meaningful results
    - Isolate serial kernels
  - Keep a working version of the code!
    - Getting the wrong answer faster is not the goal.
  - Optimize on the architecture on which you intend to run
    - Optimizations for one architecture will not necessarily translate
  - The compiler is your friend!
    - If you find yourself coding in machine language, you are doing to much.

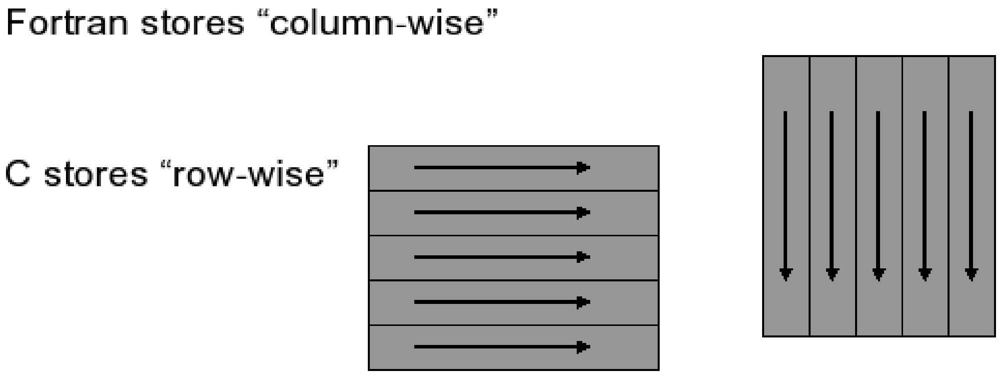
Manual Optimization Techniques

Optimization Techniques

- There are basically two different categories:
  - Improve memory performance (taking advantage of locality)
    - Better memory access patterns
    - Optimal usage of cache lines
    - Re-use of cached data
  - Improve CPU performance
    - Reduce flop count
    - Better instruction scheduling
    - Use optimal instruction set
  - A word about compilers
    - Most compilers will do many of the techniques below automatically, but is still important to understand these.

Optimization Techniques for Memory

- Stride
  - Contiguous blocks of memory
- Accessing memory in stride greatly enhances the performance



Array indexing

- There are several ways to index arrays:

```
Do j=1,M
    Do i=1,N
      ..A(i, j)
    END DO
END DO                Direct
```

```
Do j=1,M
    Do i=1,N
      ..A(i+(j-1)*N)
    END DO
END DO                Explicit
```

```
Do j=1,M
    Do i=1,N
        k=k+1
      ..A(k)
    END DO
END DO                Loop carried
```

```
Do j=1,M
    Do i=1,N
      ..A(index(i,j))..
    END DO
END DO                Indirect
```

Example (stride)

File Edit Options Buffers Tools C Cscope Help

```c
    }
    begin = clock();
    for(i=0;i<N;i++){
      for(j=0;j<N;j++){
        d[i][j] = b[j][i] + c[j][i];
      }
    }
    end = clock();
    printf("\nLoop out-stride time = %20.10lf seconds\n",(end-begin)/(CLOCKS_PER_SEC));
    begin = clock();
    for(i=0;i<N;i++){
      for(j=0;j<N;j++){
        d[i][j] = b[i][j] + c[i][j];
      }
    }
    end = clock();
    printf("\nLoop in-stride time = %20.10lf seconds\n",(end-begin)/(CLOCKS_PER_SEC));
    return 0;
}
```

--:-- stride.c (C

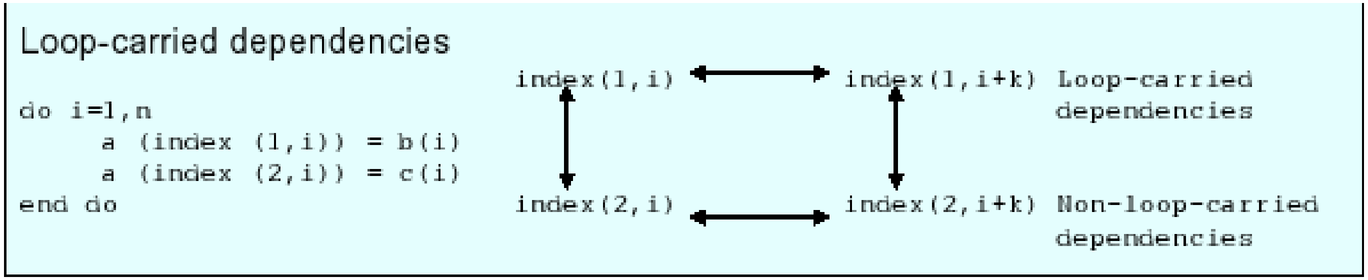
Session Edit View Bookmarks Settings Help

```
megatron:~/programming> gcc -O3 stride.c -o stride
megatron:~/programming> ./stride

Loop out-stride time =        7.3100000000 seconds

Loop in-stride time =        0.5100000000 seconds
megatron:~/programming>
```

Shell | Shell No. 2

Data Dependencies

- In order to perform hand optimization, you really need to get a handle on the data dependencies of your loops.
  - Operations that do not share data dependencies can be performed in tandum.



    - Automatically determining data dependencies is tough for the compiler.
    - great opportunity for hand optimization

Loop Interchange

- Basic idea: change the order of data independent nested loops.
- *Advantages:*
  - Better memory access patterns (leading to improved cache and memory usage)
  - Elimination of data dependencies (to increase opportunity for CPU optimization and parallelization
- *Disadvantage:*
  - Make make a short loop innermost

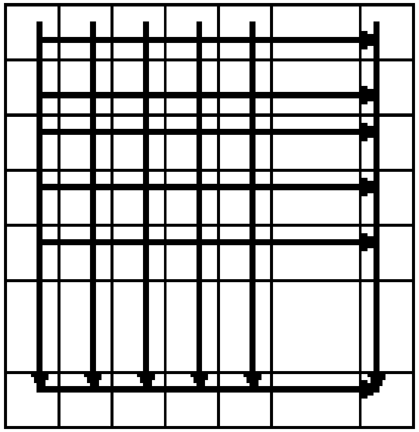
Loop Interchange – Example

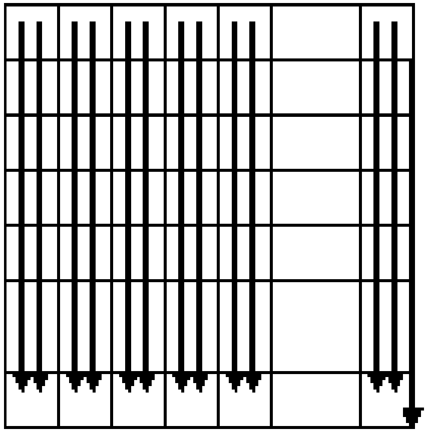
Original

```
DO i=1,N
   DO j=1,M
     C(i,j)=A(i,j)+B(i,j)
   END DO
END O
```

Interchanged loops

```
DO j=1,M
   DO i=1,N
     C(i,j)=A(i,j)+B(i,j)
   END DO
END DO
```





Access order

Storage order

Loop Interchange in C/C++

In C, the situation is exactly the opposite

interchange

```
for (j=0; j<M; j++)
    for (i=0; i<N; i++)
        C[i][j] = A[i][j] +B[i][j];
```

index reversal

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        C[i][j] = A[i][j] +B[i][j];
```

```
for (j=0; j<M; j++)
    for (i=0; i<N; i++)
        C[j][i] = A[j][i] +B[j][i];
```

- The performance benefit is the same in this case
- In many practical situations, loop interchange is much easier to achieve than index reversal

Loop Interchange – Example 2

```
DO i=1,300
   DO j=1,300
      DO k=1,300
         A (i,j,k) = A (i,j,k)+ B (i,j,k)* C (i,j,k)
      END DO
   END DO
END DO
```

Loop order	x335 (P4 2.4Ghz)	x330 (P3 1.4Ghz)
i j k	8.77	9.06
i k j	7.61	6.82
j i k	2	2.66
j k i	0.57	1.32
k i j	0.9	1.95
k j i	0.44	1.25

Compiler Loop Interchange

- GNU compilers:
  - -floop-interchange
- PGI compilers:
  - -Mvect Enable vectorization, including loop interchange
- Intel compilers:
  - -O3 Enable aggressive optimization, including loop transformations

CAUTION: Make sure thaour program still works after this!

Loop Unrolling

- Computation cheap... branching expensive
  - Loops, conditionals, etc. Cause branching instructions to be performed.
  - Looking at a loop...

```
for( i = 0; i < N; i++){
    do work....
}
```

Every time this statement is hit, a branching instruction is called.

More work, less branches

So optimizing a loop would involve increasing the work per loop iteration.

Loop unrolling

- Normal loop

```
do i=1,N
   a(i)=b(i)+x*c(i)
enddo
```

- Manually unrolled loop

```
do i=1,N,4
   a(i)=b(i)+x*c(i)
   a(i+1)=b(i+1)+x*c(i+1)
   a(i+2)=b(i+2)+x*c(i+2)
   a(i+3)=b(i+3)+x*c(i+3)
enddo
```

- Good news – compilers can do this in the most helpful cases
- Bad news – compilers sometimes do this where it is not helpful and or valid.
- This is not helpful when the work inside the loop is not mostly number crunching.

Loop Unrolling - Compiler

GNU compilers:

-funrollloops	Enable loop unrolling
-funrollallloops	Unroll all loops; not recommended

PGI compilers:

-Munroll	Enable loop unrolling
-Munroll=c:N	Unroll loops with trip counts of at least **N**
-Munroll=n:M	Unroll loops up to **M** times

Intel compilers:

-unroll	Enable loop unrolling
-unrollM	Unroll loops up to **M** times

CAUTION: Make sure that your program still works after this!

Loop Unrolling Directives

```
program dirunroll
integer,parameter :: N=1000000
real,dimension(N):: a,b,c
real:: begin,end
real,dimension(2):: rtime
common/saver/a,b,c
call random_number(b)
call random_number(c)
x=2.5
begin=dtime(rtime)
!DIR$ UNROLL 4
do i=1,N
a(i)=b(i)+x*c(i)
end do
end=dtime(rtime)
print *,' my loop time (s) is
',(end)
flop=(2.0*N)/(end)*1.0e6
print *,' loop runs at ',flop,'
MFLOP'
print *,a(1),b(1),c(1)
end
```

- Directives provide a very portable way for the compiler to perform automatic loop unrolling.
  - Compiler can choose to ignore it.

Blocking for cache (tiling)

- Blocking for cache is
  - An optimization that applies for datasets that do not fit entirely into cache
  - A way to increase spatial locality of reference i.e. exploit full cache lines
  - A way to increase temporal locality of reference i.e. improves data reuse
- Example, the transposing of a matrix

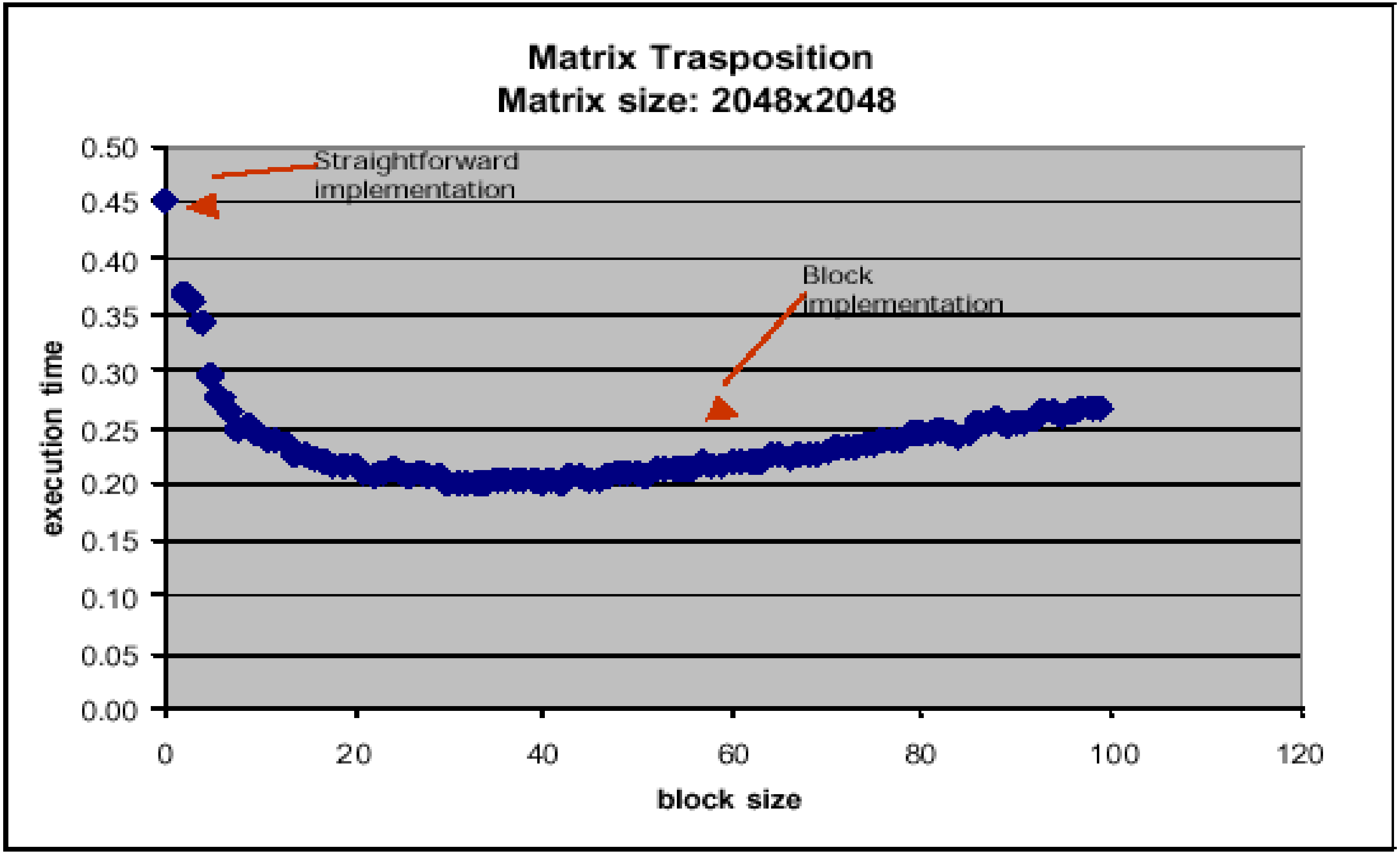
```
do i=1,n
  do j=1,n
     a(i,j)=b(j,i)
  end do
end do
```

Block algorithm for transposing a matrix

- block data size = bsize
  - mb = n/bsize
  - nb = n/bsize
- These sizes can be manipulated to coincide with actual cache sizes on individual architectures

```
do ib = 1, nb
 ioff = (ib-1) * bsiz
 do jb = 1, mb
   joff = (jb-1) * bsiz
    do j = 1, bsiz
      do i = 1, bsiz
        buf(i,j) = x(i+ioff, j+joff)
     enddo
    enddo
    do j = 1, bsiz
      do i = 1, j-1
       bswp = buf(i,j)
       buf(i,j) = buf(j,i)
       buf(j,i) = bswp
      enddo
     enddo
    do i=1,bsiz
      do j=1,bsiz
       y(j+joff, i+ioff) = buf(j,i)
      enddo
    enddo
  enddo
enddo
```

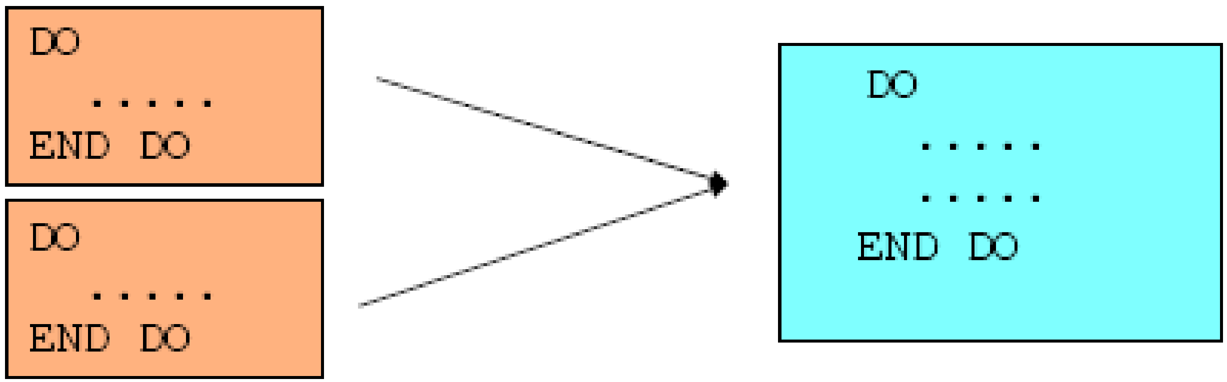
Results...



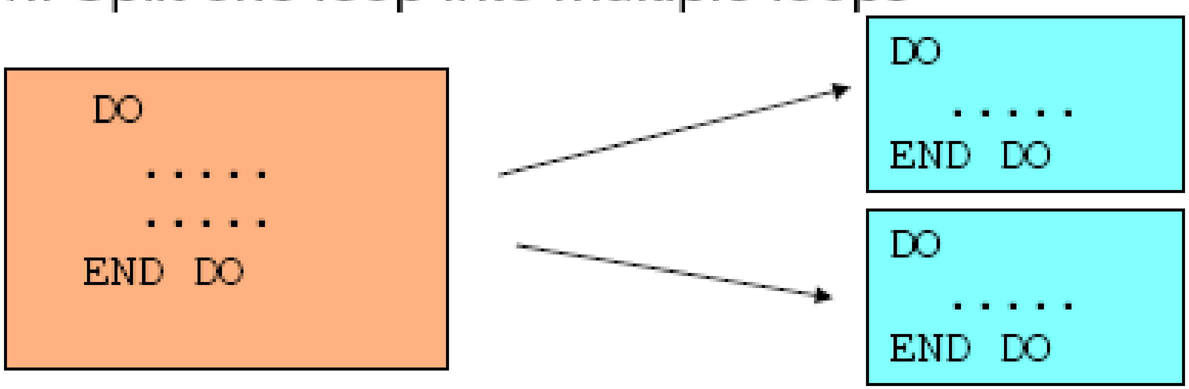
Matrix Trasposition
Matrix size: 2048x2048
Straightforward implementation
Block implementation
execution time
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20
40
60
80
100
120
block size

Loop Fusion and Fission

Fusion: Merge multiple loops into one



Fission: Split one loop into multiple loops



Loop Fusion Example

```
DO i=1,N
   B(i)=2*A(i)
END DO
```

```
DO k=1,N
   C(k)=B(k)+D(k)
END DO
```

```
DO ii=1,N
   B(ii)=2*A(ii)
   C(ii)=B(ii)+D(ii)
END DO
```

Potential for Fusion: dependent operations in separate loops

Advantage:

- Re-usage of array B()

Disadvantages:

- In total 4 arrays now contend for cache space
- More registers needed

Loop Fission Example

```
DO ii=1,N
   B(i)=2*A(i)
   D(i)=D(i-1)+C(i)
END DO
```

```
DO ii=1,N
   B(ii)=2*A(ii)
END DO
```

```
DO ii=1,N
   D(ii)=D(ii-1)+C(ii)
END DO
```

Potential for Fission: independent operations in a single loop

Advantage:

- First loop can be scheduled more efficiently and be parallelised as well

Disadvantages:

- Less opportunity for out-of-order superscalar execution
- Additional loop created (a minor disadvantage)

Prefetching

- Modern CPU's can perform anticipated memory lookups ahead of their use for computation.
  - Hides memory latency and overlaps computation
  - Minimizes memory lookup times
- This is a very architecture specific item
- Very helpful for regular, in-stride memory patterns

GNU:	
-fprefetch-loop-arrays	If supported by the target machine, generate instructions to prefetch memory to improve the performance of loops that access large arrays.
PGI:	
-Mprefetch[=option:n] **-Mnoprefetch**	Add (don't add) prefetch instructions for those processors that support them (Pentium 4,Opteron); -Mprefetch is default on Opteron; -Mnoprefetch is default on other processors.
Intel:	
-O3	Enable -O2 optimizations and in addition, enable more aggressive optimizations such as loop and memory access transformation, and prefetching.

Optimizing Floating Point performance

- Operation replacement
  - Replacing individual time consuming operations with faster ones
  - Floating point division
    - Notoriously slow, implemented with a series of instructions
    - So does that mean we cannot do any division if we want performance?
  - IEEE standard dictates that the division must be carried out
    - We can relax this and replace the division with multiplication by a reciprocal
    - Compiler level optimization, rarely helps doing this by hand.
    - Much more efficient in machine language than straight division, because it can be done with approximates

IEEE relaxation

```
GNU:
-funsafe-math-optimizations
    Allow optimizations for floating-point arithmetic that (a) assume
that arguments and results are valid and (b) may violate IEEE or ANSI
standards.

PGI:
--Kieee -Knoieee (default)
    Perform floating-point operations in strict conformance with the
IEEE 754 standard.  Some optimizations are disabled with -Kieee, and a
more accurate math library is used.  The default -Knoieee uses faster
but very slightly less accurate methods.

INTEL:
--no-prec-div  (i32 and i32em)
    Enables  optimizations  that  give  slightly  less precise results
than full IEEE division. With some optimizations, such as -xN and -xB,
the compiler may change floating-point  division  computations  into
multiplication by the reciprocal of the denominator.
```

Keep in mind! This does reduce the precision of the math!

Elimination of Reduntant Work

- Consider the following piece of code

```
do j = 1,N
        do i = 1,N
                A(j) = A(j) + C(i,j)/B(j)
        enddo
enddo
```

It is clear that the division by B(j) is redundant and can be pulled out of the loop

```
do j = 1,N
        sum = 0.0D0
        do i = 1,N
                sum = sum + C(i,j)
        enddo
        A(j) = A(j) + sum/B(j)
enddo
```

Elimination of Redundant Work

```
do k = 1,N
       do j = 1,N
              do i = 1,N
                     A(k) = B(k) + C(j) + D(i)
              enddo
       enddo
enddo
```

Array lookups cost time

By introducing constants and precomputing values, we eliminate a bunch of unnecessary fops

This is the type of thing compilers can do quite easily.

```
do k = 1,N
       Bk = B(k)
       do j = 1,N
              BkCj = Bk + C(j)
              do i = 1,N
                     A(k) = BkCj + D(i)
              enddo
       enddo
enddo
```

Function (Procedure) Inlining

- Calling functions and subroutines requires overhead by the CPU to perform
  - The instructions need to be looked up in memory, the arguments translated, etc..
- Inlining is the process by which the compiler can replace a function call in the object with the source code
  - It would be like creating your application in one big function-less format.
- Advantage
  - Increase optimization opportunities
  - Particularly advantegeous (necessary) when a function is called a lot, and does very little work (e.g. max and min functions).
- Particularly important in C++!!!

Function (Procedure) Inlining Compiler Options

GNU compilers:

`-fno-inline`	Disable inlining
`-finline-functions`	Enable inlining of functions

PGI compilers:

`-Mextract=option[,option,…]`	Extract functions selected by `option` for use in inlining; `option` may be `name:function` or `size:N` where `N` is a number of statements
`-Minline=option[,option,…]`	Perform inlining using `option`; `option` may be `lib:filename.ext`, `name:function`, `size:N`, or `levels:P`

Intel compilers:

`-ip`	Enable single-file interprocedural optimization, including enhanced inlining
`-ipo`	Enable interprocedural optimization across files

In source

- You can use inline directives to specify that you want a function inlined:

```
inline int fun2() __attribute__((always_inline));
inline int fun2() { return 4 + 5; }
```

- You can find out if functions have been inlined properly, the code nm can be looked at.
  - If the function is not in the nm output, it has been inlined.
- Inlining can cause a function to no longer be accessible by a debugger.

Superscalar Processors

- Processors which have multiple functional units are called superscalar (instruction level parallelism)
- Examples:
  - All modern processors
  - All can do multiple floating point and integer procedures in one clock cycle
- Special instructions
  - SSE (Streaming SIMD Extensions)
    - Allow users to take advantage of this power by packing mutliple operations into one register.
    - SSE2 for double-precision
    - Right now, 4 way is very common (Intel Corei7), but 16-way on the horizon.
    - Intel PHI is an extereme form of this.
    - Much much more difficult to get peak performance.

Instruction Set Extension Compiler Options

```
GNU:
  -mmmx/no-mmx              These switches enable or disable the use of
                            built-in functions that allow direct access to
  -msse                     the MMX, SSE, SSE2, SSE3 and 3Dnow
  -mno-sse                  extensions of the instruction set
  -msse2 /  -mno-sse2
  -msse3 /  -mno-sse3
  -m3dnow / -mno-3dnow

PGI:
  --fastsse
     Chooses generally optimal flags for a processor that supports
     SSE instructions (Pentium 3/4, AthlonXP/MP, Opteron) and SSE2
     (Pentium 4, Opteron).  Use pgf90 -fastsse -help to see the
     equivalent switches.

INTEL:
  -arch SSE   Optimizes  for Intel Pentium 4 processors with Streaming
              SIMD Extensions (SSE).
  -arch SSE2  Optimizes for Intel Pentium 4 processors with Streaming
               SIMD Extensions 2 (SSE2).
```

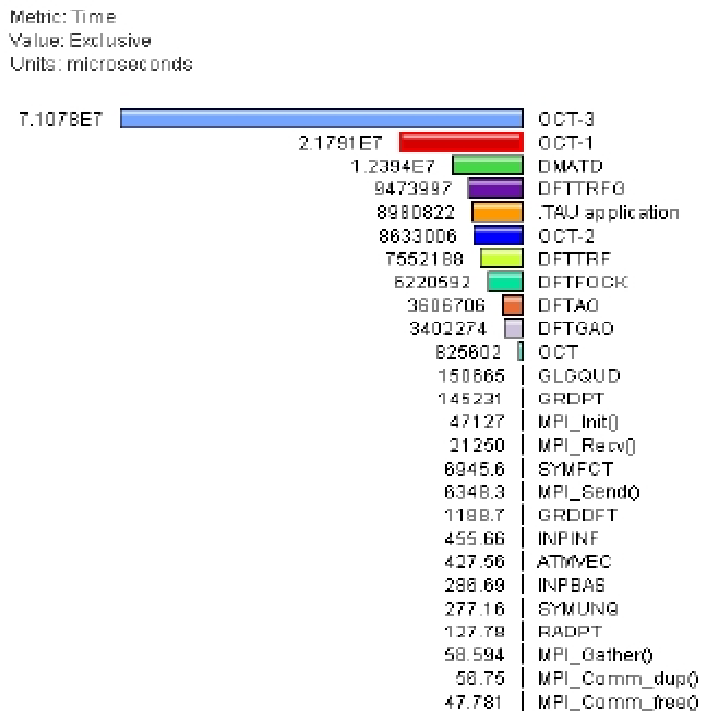
How do you know what the compiler is doing?

- Compiler Reports and Listings
  - By default, compilers don't say much unless you screwed up.
  - One can generate optimization reports and listing files to yeild output that shows what optimizations are performed

GNU compilers	None
PGI compilers	
`-Minfo=option[,option,...]`	Prints information to `stderr` on `option`; option can be one or more of **time, loop, inline, sym, or all**
`-Mneginfo=option[,option]`	Prints information to `stderr` on why optimizations of type `option` were not performed; option can be **concur or loop**
`-Mlist`	Generates a listing file
Intel compilers	
`-opt_report`	Generates an optimization report on stderr
`-opt_report_file filename`	Generates an optimization report to `filename`

Case Study: GAMESS

- Mission from the DoD – Optimize GAMESS DFT code on an SGI Altix
- First step: profile the code



Case Study: GAMESS

- **Before**

  Source code from the OCT subroutine from the GAMESS program. This portion of code is represented in the loop level profiling in the previous slide by the OCT-3 moniker.

```
DO K=1,NITR
   F4=F4*(1.5D+00-0.5D+00*F4*F4)
END DO
F2=0.5D+00*F4
```

- **After**

  Optimized source code from the OCT subroutine from the GAMESS program.

```
F41 =   F4*(1.5D0-0.5D0*F4*F4)
F42 = F41*(1.5D0-0.5D0*F41*F41)
F43 = F42*(1.5D0-0.5D0*F42*F42)
F44 = F43*(1.5D0-0.5D0*F43*F43)
F2 = 0.5D0*F44
```

- **New code is 5x faster through this section of the program**

- Further inspection of the Itanium archtecture showed 2 things:
  - The compilers were really bad at loop optimization
  - The overhead for conditionals is enormous

Take Home Messages…

- Performance programming on single processors requires
  - Understanding memory
    - levels, costs, sizes
  - Understand SSE and how to get it to work
    - In the future this will one of the most important aspects of processor performance.
  - Understand your program
    - No subsitute for speding quality time with your code.
- Do not spend a lot of time doing what I compiler will do automatically.
  - Start with compiler optimizations!
- Code optimization is hard work!
  - We haven't even talked about parallel applications yet!