Outline

1. Introduction
2. Compiler Options
3. Performance Libraries
4. Code Optimizations
1 General Optimization Procedure

Optimization in code design/development:

- Requires understanding of common architecture features
- Requires sense of how compilers map code to instructions.

Optimization is an iterative process:

- Profile code
- Work on most time intensive blocks
- Repeat
1 Compiler Options

• Three important Categories
  – Optimization Level
  – Architecture Specification
  – Interprocedural Optimization

You should always have at least one option from each category!
2 Compilers and Optimization

• Compilers can perform significant optimization
  – The compiler follows your lead!
  – Structure code to make apparent what the compiler should do (so that
    the compilers and others can understand it).
  – Use simple language constructs (e.g. don’t use pointers, or OO code).

• Use latest compilers.
  – Always check compiler options
    <compiler-command> --help  {lists/explains options}
  – Look for architecture options for your system
    See User Guides – usually lists “best practice” options
    cat /proc/cpuinfo  {shows cpu information}

• Experiment with different options.
• May need routine-specific options (use -ipo).
Optimization Level: –On

• -O0 no optimization: Fast compilation, disables optimization
• -O1 optimize for speed, but disables optimizations which increase code size
• -O2 default optimization
• -O3 aggressive optimization: rearrangement of code, i.e. scalar replacement, loop transformation. Compile time/space intensive and/or marginal effectiveness; may change code semantics and results (sometimes even breaks codes!)
Optimization Levels

• Operations performed at default optimization level
  – instruction rescheduling
  – copy propagation
  – software pipelining
  – common subexpression elimination
  – prefetching, (some loop transformations)

• Operations performed at aggressive optimization levels
  – Usually enabled by –O3
  – more aggressive prefetching, loop transformations
Architecture Specification

X87 instruction sets are now replaced by SSE “Vector” instruction sets.
(S)SSE = (Supplemental) Streaming SIMD Extension
SSE instructions sets are chip dependent

(SSE instructions pipeline and simultaneously execute independent operations to get multiple results per clock period.)
The –x<codes> { code = W, P, T, O, S} directs the compiler to use most advanced SSE instruction set for the target hardware.
Architecture Specification

**Intel** (SSSE is for Intel chips only!)

Processor-specific optimization options (all do SSE and SSE2):
- `-xT` includes SSE3 & SSSE3 instructions for EM64T (Lonestar, v. 10.1)
- `-xW` **no supplemental** Instructions (Ranger, v. 10.1)
- `-xO` includes SSE3 Instructions (Ranger, v. 10.1)

**PGI**
- `-tp barcelona-64` uses instruction set for barcelona chip
Interprocedural Optimization (IP)

- Most compilers will handle IP within a single file (option –ip)
- The Intel -ipo compiler option does more
  - It adds additional information to each object file.
  - Then, during loading, the code is recompiled and IP among ALL objects is performed.
  - May take much more time: Code is recompiled during linking
  - It is **Important** to include options in link command (-ipo –O# -xW, etc.)
    (special Intel xild loader replaces ld)
  - When archiving in a library, you must use xiar, instead of ar.
Interprocedural Optimization (IP)

**Intel**

- `-ip` enable single-file interprocedural (IP) optimizations (within files). Line numbers produced for debugging
- `-ipo` enable multi-file IP optimizations (between files)

**PGI**

- `-Mipa=fast,inline` Interprocedural Optimization
Other Intel Compiler Options

Other options:
- **-g** debugging information, generates symbol table
- **-vec_report[#]** {#=0-5}, controls vector diagnostic reporting
- **-C** enable extensive runtime error checking (-CA, -CB, -CS, -CU, -CV)
- **-convert <kwd>** specify file format
  - keyword: big_endian, cray, ibm, little_endian, native, vaxd
- **-openmp** enable the parallelizer to generate multi-threaded code based on the OpenMP directives.
- **-openmp_report** controls level of diagnostic reporting
- **-static** create a static executable for serial applications. MPI applications compiled on Lonestar cannot be built statically.
Other PGI Compiler Options

Processor-specific optimization options:

- **-fast**
  -O2 -Munroll=c:1 -Mnoframe -MIre -Mautoinline
  -Mvect=sse -Mscalarsse -Mcache_align -Mflushz

- **-mp**
  thread generation for OpenMP directives

- **-Minfo=mp,ipa**
  OpenMP/Interprocedural Opt. reporting
Compilers - Best Practice

- Normal compiling for Ranger

  intel  `icc/ifort -O3 -ipo -xW prog.c/cc/f90`
  pgi  `pgcc/pgcpp/pgf95 -fast -tp barcelona-64 -Mipa=fast,inline`
  prog.c/cc/f90
  gnu  `gcc -O3 -fast -xipo -mtune=barcelona -march=barcelona prog.c`

- O2 is default opt, compile with –O0 if this breaks (very rare)

- The effects of -xW and -xO options may vary

- Don’t include debug options for a production compile!
  `ifort -O2 -g -CB test.c`
3 Performance Libraries

- Optimized for specific architectures
- Use library routines instead of hand-coding your own
- In “hot spots”, never write library functions by hand.
- Offered by different vendors (ESSL/PESSL on IBM systems, Intel MKL for x86-64, AMD ACML, Cray libsci for Cray systems, SCSL for SGI)
- Numerical Recipes books DO NOT provide optimized code. (Libraries can be 100x faster).
<table>
<thead>
<tr>
<th>Performance</th>
<th>Math Libs</th>
<th>Method Libs</th>
<th>Applications</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>gprof</td>
<td>SPRNG</td>
<td>PETSc</td>
<td>Amber</td>
<td>NetCDF</td>
</tr>
<tr>
<td>TAU</td>
<td>Metis/parmetis</td>
<td>PLAPACK</td>
<td>NAMD</td>
<td>HDF (4/5)</td>
</tr>
<tr>
<td>PAPI</td>
<td>FFTW (2/3)</td>
<td>SCALAPACK</td>
<td>GROMACS</td>
<td>Parallel</td>
</tr>
<tr>
<td>DDT</td>
<td>MKL</td>
<td>SLEPc</td>
<td>Gamess</td>
<td>I/O</td>
</tr>
<tr>
<td>...</td>
<td>GSL</td>
<td>...</td>
<td>NWchem</td>
<td>GridFTP</td>
</tr>
<tr>
<td></td>
<td>GotoBLAS</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Intel MKL 10.0 (Math Kernel Library)

- Optimized for the IA32, x86-64, IA64 architectures
- supports both Fortran and C interfaces
- Includes functions in the following areas:
  - BLAS (levels 1-3)
  - LAPACK
  - FFT routines
  - … others
  - Vector Math Library (VML)
Intel MKL 10.0 (Math Kernel Library)

- Enabling MKL
  - module load mkl
  - module help mkl

- Example Compile

  mpicc -I$TACC_MKL_INC mkl_test.c -L$TACC_MKL_LIB -lmkl_<>
  mpiif90 mkl_test.f90 -L$TACC_MKL_LIB -lmkl_<>
Code Optimization

• **Always minimize stride length**
  – Stride length 1 is optimal for vectorizable code.
  – This increases cache efficiency, and sets up hardware and software prefetching.
  – Stride lengths of powers of two are typically the worst case scenario leading to cache misses.

• **Strive to write Vectorizable Loops**
  – Can be sent to a SIMD Unit
  – Can be unrolled and pipelined
  – Can be parallelized through OpenMP Directives
  – Can be “automatically” parallelized (be careful…)

G4/5
Intel/AMD
Cray

Velocity Engine (SIMD)
MMX, SSE, SSE2, SSE3 (SIMD)
Vector Units
SIMD (Single Instruction Multiple Data)

SSE (Streaming SIMD Extensions) instructions operate on multiple data arguments simultaneously

4 Code Optimization

- Write loops with independent iterations, so that SSE instructions can be employed
Approx. Memory Bandwidths & Sizes

Relative Memory Bandwidths
- Functional Units
  - Registers
  - L1 Cache: ~50 GB/s
  - L2 Cache: ~25 GB/s
  - L3 Cache Off Die: ~12 GB/s
  - Local Memory: ~8 GB/s

Relative Memory Sizes
- L1 Cache: 16/32 KB
- L2 Cache: 1 MB
- Memory: 1 GB

Latency
- Processor: ~5 CP
- L1 Cache: ~15 CP
- L2 Cache: ~300 CP
Code Optimization

When is Inlining important?
   When the function is a hot spot
   When the call-overhead to work ratio is high
   When it can benefit from Interprocedural Optimization

The C inline keyword provides inlining within source.
As you develop “think inlining”.
Use –ip or –ipo to allow the compiler to inline.
Example: procedure inlining

```fortran
program MAIN
  integer :: ndim=2, niter=10000000
  real*8 :: x(ndim), x0(ndim), r
  integer :: i, j
  ...
  do i=1,100000
    ...
    r=dist(x,x0,ndim)
    ...
  end do
  ...
end program

real*8 function dist(x,x0,n)
real*8 :: x0(n), x(n), r
integer :: j,n
r=0.0
do j=1,n
  r=r+(x(j)-x0(j))**2
end do
dist=r
end function
```

function `dist` is called \textit{niter} times

```fortran
program MAIN
  integer, parameter :: ndim=2
  real*8 :: x(ndim), x0(ndim), r
  integer :: i, j
  ...
  do i=1,100000
    ...
    r=0.0
    do j=1,ndim
      r=r+(x(j)-x0(j))**2
    end do
    ...
  end do
end program
```

Loop \textit{j} is called \textit{niter} times
The following snippets of code illustrate the correct way to access contiguous elements. i.e. stride 1, for a matrix in both C and Fortran.

Fortran Example:

```fortran
real*8 :: a(m,n), b(m,n), c(m,n)
...
do i=1,n
   do j=1,m
      a(j,i)=b(j,i)+c(j,i)
   end do
end do
```

C Example:

```c
double a[m][n], b[m][n], c[m][n];
...
for (i=0;i < m;i++){
   for (j=0;j < n;j++){
      a[i][j]=b[i][j]+c[i][j];
   }
}
```
Code Optimization

• Also, for large and small arrays, always try to arrange data so that structures are arrays with a unit (1) stride.

Bandwidth Performance Code:

```
do i = 1,10000000,istride
  sum = sum + data(i)
end do
```
Code Optimization

Loop interchange can help in the case of a DAXPY loop:

```fortran
integer, parameter :: nkb=16, kb=1024, n=nkb*kb/8
real*8 :: x(n), a(n,n), y(n)
...
do i=1,n
    s=0.0
    do j=1,n
        s=s+a(i,j)*x(j)
    end do
    y(i)=s
end do
```

```fortran
integer, parameter :: nkb=16, kb=1024, n=nkb*kb/8
Real*8 :: x(n), a(n,n), y(n)
...
do j=1,n
    do i=1,n
        y(i)=y(i)+a(i,j)*x(j)
    end do
end do
```
The objective of array blocking is to work with small array blocks when expressions contain mixed-stride operations. It uses complete cache lines when they are brought in from memory, and hence avoid possible eviction that would otherwise ensue without blocking.

\[
\begin{align*}
    \text{do } & i=1,n \\
    \text{do } & j=1,n \\
    & A(j,i)=B(i,j) \\
\end{align*}
\]

\[
\begin{align*}
    \text{do } & i=1,n,2 \\
    \text{do } & j=1,n,2 \\
    & A(j,i)=B(i,j) \\
    & A(j+1,i)=B(i+1,j) \\
    & A(j,i+1)=B(i,j+1) \\
    & A(j+1,i+1)=B(i+1,j+1) \\
\end{align*}
\]
Code Optimization

Array Blocking

```fortran
real*8 a(n,n), b(n,n), c(n,n)
do ii=1,n,nb
    do jj=1,n,nb
        do kk=1,n,nb
            do i=ii,min(n,ii+nb-1)
                do j=jj,min(n,jj+nb-1)
                    do k=kk,min(n,kk+nb-1)
                        c(i,j)=c(i,j)+a(j,k)*b(k,i)
                    end do
                end do
            end do
        end do
    end do
end do
```

Much more efficient implementations exist, in HPC scientific libraries (ESSL, MKL, ACML,...).
Code Optimization

Even low-stride is effective when accessing data in cache.

Bandwidth Performance Code (assume data is in cache):

```fortran
    do i = 1,50000,istride
        sum = sum + data(i)
    end do
```
Code Optimization

In some cases, an entire loop can be replaced with a single call to a vector function. For example, the loop below can be written as a call to vdInvSqrt in the Intel VML:

```c
for (i=0;i<n;i++) {
    y[i]=1.0/sqrt(x[i]);
}
```

```c
for (i=0;i<n;i++) {
    y[i]=a*sin(x[i]) + b*cos(x[i]);
}
```

vdInvSqrt(n,x,y);
vdSinCos(n,x,s,c);
for (i=0;i<n;i++) {
    y[i]=a*s[i] + b*c[i];
}

But, how do you make something like this portable? -- “ifdef”, in C and F90.
#IFDEF example

program main
integer, Parameter :: n=100, nn=2*n, nap=nn*(nn+1)/2
real(8), Parameter :: xmax=20.0, xmin=-xmax

#ifdef _IBM
integer            :: iopt=20
integer, parameter :: naux=3*nn
real(8):: ap(nap), eval(nn), work(naux)
#else defined _IA32
integer            :: info
real(8):: ap(nap), eval(nn), work(3*nn)
#endif...
#endif

... #ifdef _IA32
   call DSPEV('n','u',nn,ap,eval,evec,nn,work,info)
#endif...
#endif...
end program
Code Optimization

Loop fusion:

Loop fusion combines two or more loops of the same iteration space (loop length) into a single loop:

```c
for (i=0; i<n; i++) {
    a[i] = x[i] + y[i];
}
for (i=0; i<n; i++) {
    b[i] = 1.0 / x[i] + z[i];
}
```

Costly (at least 30 CP)

```c
for (i=0; i<n; i++) {
    a[i] = x[i] + y[i];
    b[i] = 1.0 / x[i] + z[i];
}
```

Only n memory accesses for X array.
Five streams created.
Division many not be pipelined!
The opposite of loop fusion is loop distribution or fission. Fission splits a single loop with independent operations into multiple loops:

\[
\begin{align*}
\text{do } i=1,n \\
a(i) &= b(i) + c(i) \times d(i) \\
\text{end do} \\
\text{do } i=1,n \\
e(i) &= f(i) - g(i) \times h(i) + p(i) \\
q(i) &= r(i) + s(i) \\
\text{end do} \\
\text{do } i=1,n \\
a(i) &= b(i) + c(i) \times d(i) \\
\text{end do} \\
\text{do } i=1,n \\
e(i) &= f(i) - g(i) \times h(i) + p(i) \\
\text{end do} \\
\text{do } i=1,n \\
q(i) &= r(i) + s(i) \\
\text{end do}
\end{align*}
\]
References

• Books
  * High Performance Computing by Kevin Dowd and Charles Severance (O’Reilly book) -- general study of high performance computing
  * Performance Optimization of Numerically Intensive Codes by Stefan Goedecker and Adolfy Hoisie (Siam book, Society for Industrial and Applied Mathematics)

• TACC User Guides
  www.tacc.utexas.edu/services/userguides/ranger/
  www.tacc.utexas.edu/services/userguides/lonestar/

• Compilers

• Optimization
  http://cache-www.intel.com/cd/00/00/21/92/219281_compiler_optimization.pdf
References

• Libraries

  GotoBLAS  www.tacc.utexas.edu/resources/software/
  Dense and band matrix software (ScaLAPACK)
      www.netlib.org/scalapack
  Large sparse eigenvalue software (PARPACK and ARPACK)
      www.caam.rice.edu/software/ARPACK/