Cornell University
Center for Advanced Computing

Code Optimization

Brandon Barker

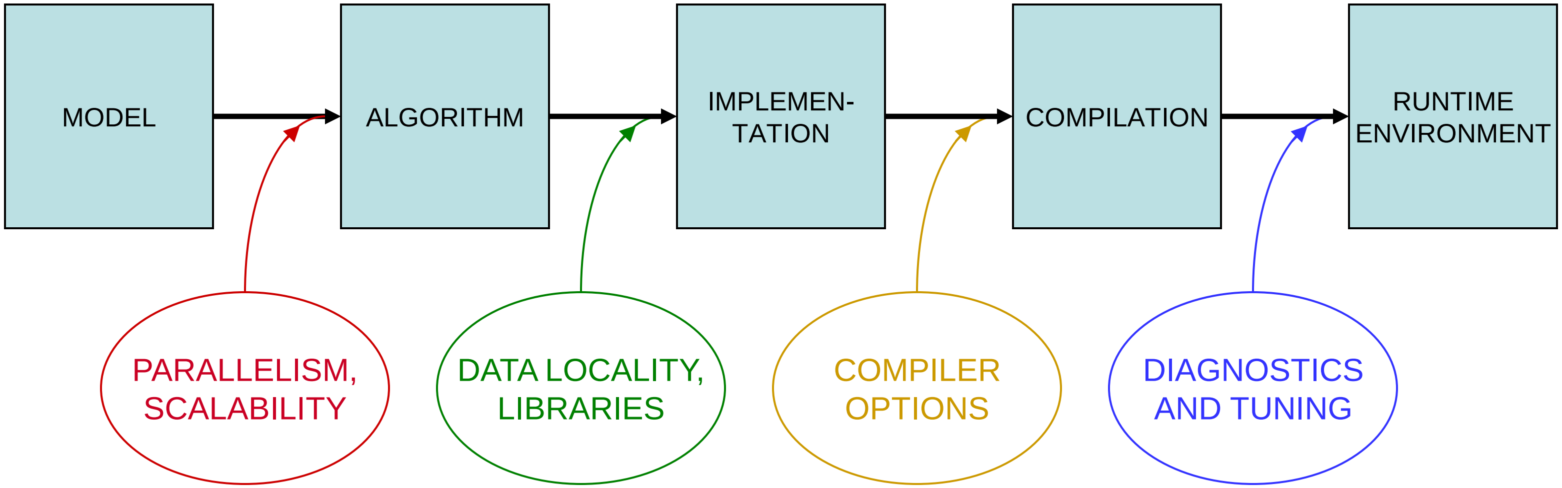
Computational Scientist

Cornell University Center for Advanced Computing (CAC) brandon.barker@cornell.edu

Workshop: High Performance Computing on Stampede

January 15, 2015

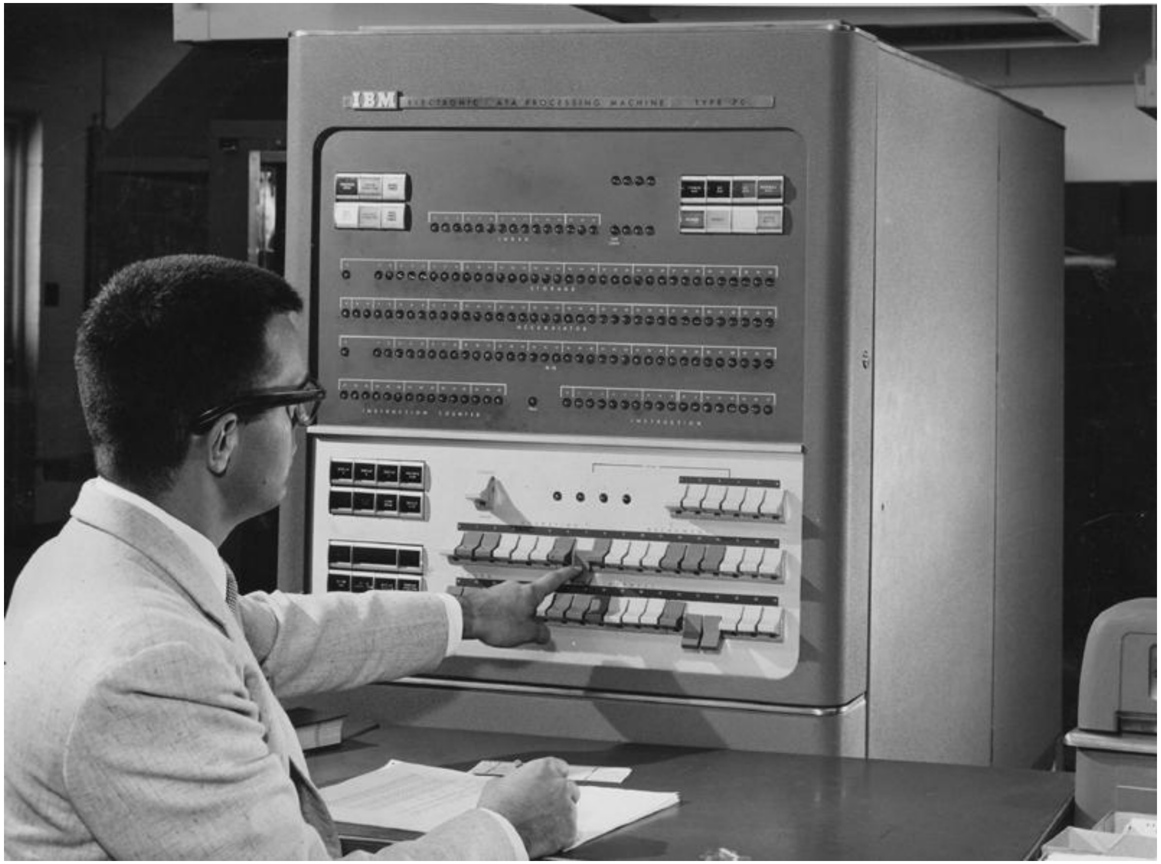
Putting Performance into Development: Libraries



Starting with how to *design* for parallelism and scalability…

…this talk is about the principles and practices during various stages of code *development* that lead to better performance on a per-core basis

In HPC, the Compiler Can't Do Everything



What Matters Most in Per-Core Performance?

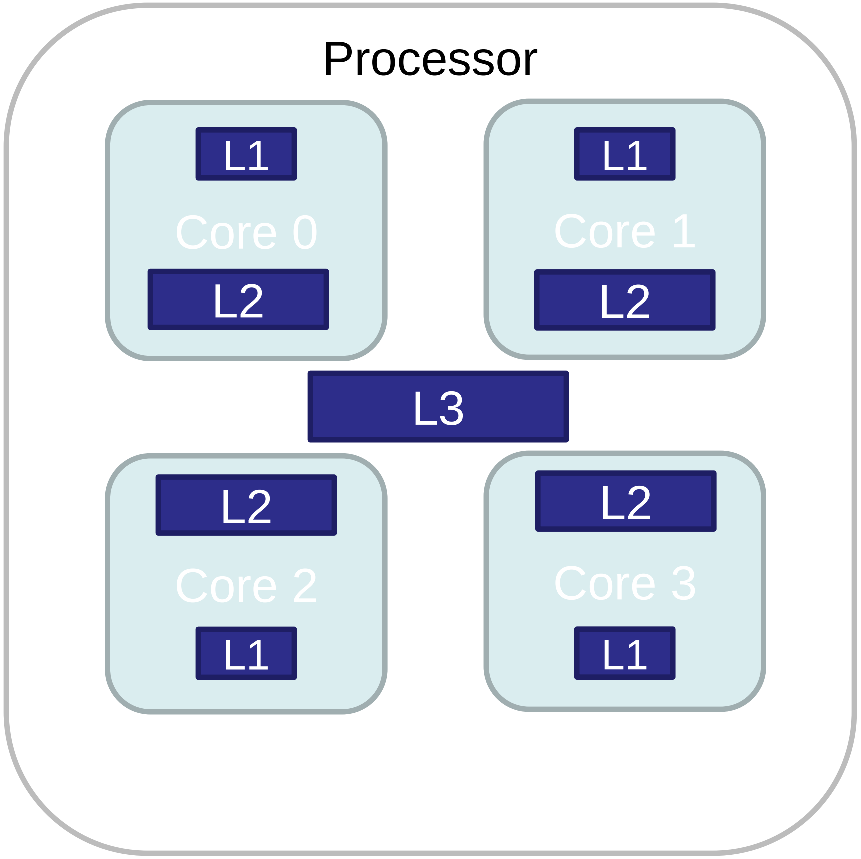
Good memory locality!

- Code should access contiguous, stride-one memory addresses
  - Memory IO (bandwidth and latency) is limiting
  - data always arrive in cache lines which include neighbors
  - loops are vectorizable via SSE, AVX
  - Align data on important boundaries; items won't straddle boundaries, so access is more efficient

- Code should emphasizes cache reuse
  - when multiple operations on a data item are grouped together, the item remains in cache, where access is much faster than from RAM

- Locality is even more important for coprocessors than it is for CPUs

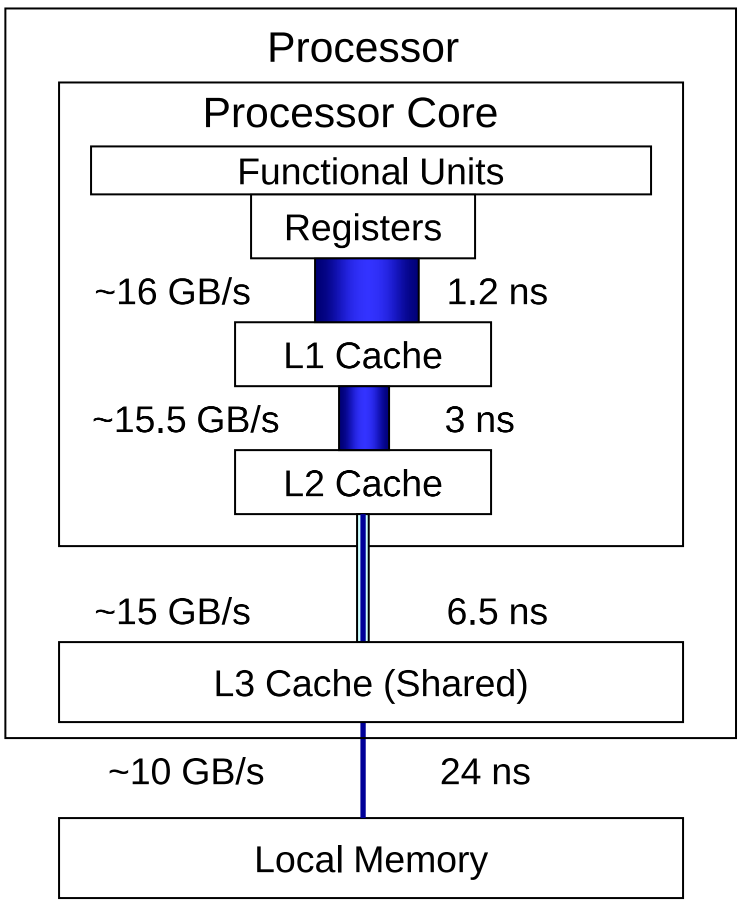
Important Aspects of Computer Architecture



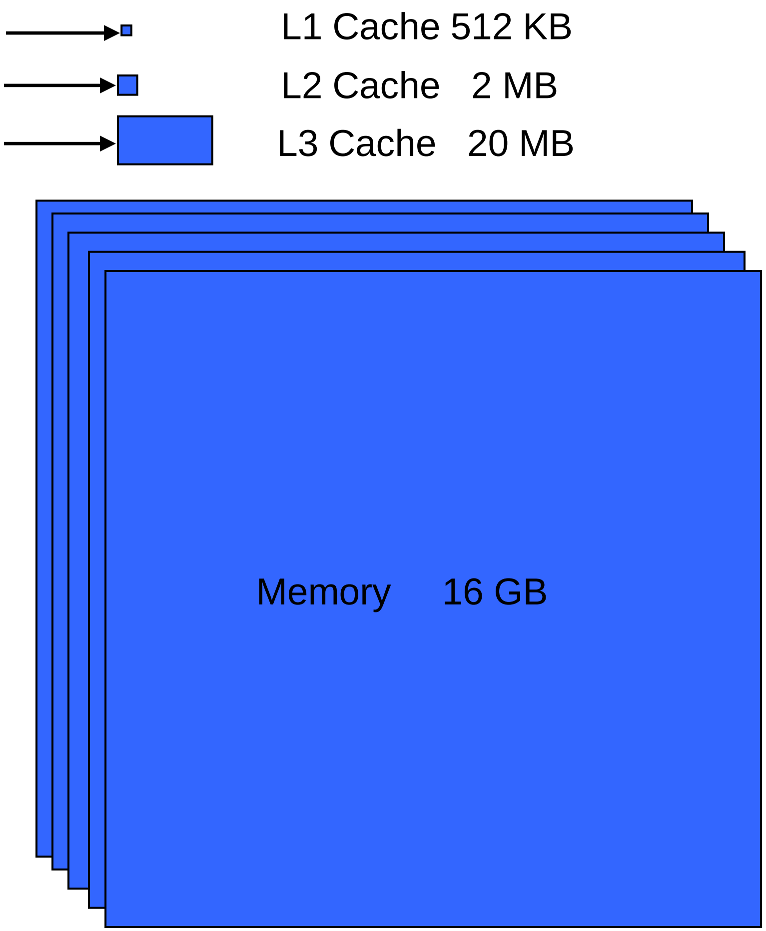
- CPU core – a processing unit that supports a thread of execution.
- Processor – all the cores and cache memory on a single chip.
- Cache – very fast on-chip memory (L1, L2, L3).
- Cache line – basic unit of contiguous memory fetched from main memory into cache.

Understanding The Memory Hierarchy

Memory Read Bandwidths (Left)
Memory Access Latency (Right)



Relative Memory Size (per socket)



Computer Architecture Matters

- Compiled code should exploit special instructions & hardware

- Intel SSE and AVX extensions access special registers & operations
  - 128-bit SSE registers can hold 4 floats/ints or 2 doubles simultaneously
  - 256-bit AVX registers were introduced with “Sandy Bridge”
  - 512-bit SIMD registers are present on the Intel MICs
  - Within these vector registers, vector operations can be applied
  - Operations are also pipelined (e.g., load > multiply > add > store)
  - Therefore, multiple results can be produced every clock cycle

Understanding SIMD and Micro-Parallelism

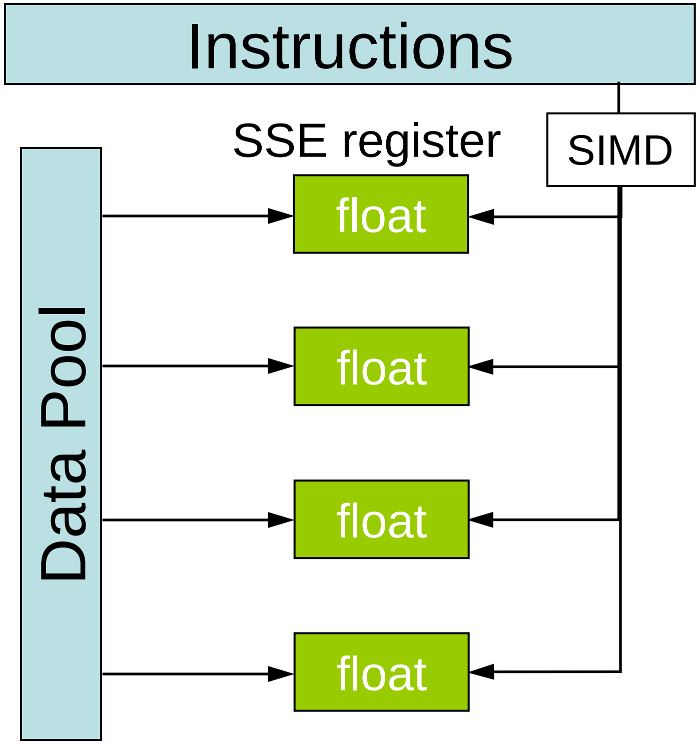
- For "vectorizable" loops with independent iterations, SSE and AVX instructions can be employed…

SIMD = *Single Instruction, Multiple Data*

SSE = *Streaming SIMD Extensions*

AVX = *Advanced Vector Extensions*

Instructions operate on multiple arguments simultaneously, in a parallel Execution Unit



Performance Libraries

- Optimized for specific architectures (chip + platform + system)
  - Take into account details of the memory hierarchy (e.g., cache sizes)
  - Exploit pertinent vector (SIMD) instructions

- Offered by different vendors for their hardware products
  - Intel Math Kernel Library (MKL)
  - AMD Core Math Library (ACML)
  - IBM ESSL/PESSL, Cray libsci, ...

- Usually far superior to hand-coded routines for “hot spots”
  - Writing your own library routines by hand is like re-inventing the wheel
  - *Numerical Recipes* books are NOT a source of optimized code: performance libraries can run 100x faster

HPC Software on Stampede, from Apps to Libs

Applications	Parallel Libs	Math Libs	Input/Output	Diagnostics
AMBER NAMD GROMACS GAMESS VASP …	PETSc ARPACK Hypre ScaLAPACK SLEPc METIS ParMETIS SPRNG …	MKL Open BLAS FFTW(2/3) GSL GLPK NumPy …	HDF5 PHDF5 NetCDF pNetCDF Parallel I/O GridFTP …	TAU PAPI …

Intel MKL 13 (Math Kernel Library)

- Accompanies the Intel 13 compilers
- Optimized by Intel for all current Intel architectures
- Supports Fortran, C, C++ interfaces
- Includes functions in the following areas:
  - Basic Linear Algebra Subroutines, for BLAS levels 1-3
  - LAPACK, for linear solvers and eigensystems analysis
  - Fast Fourier Transform (FFT) routines
  - Transcendental functions
  - Vector Math Library (VML) for vectorized transcendentals
- Incorporates shared- and distributed-memory parallelism
  - OpenMP multithreading is built in, just set OMP_NUM_THREADS > 1
  - Link with BLACS to provide optimized ScaLAPACK

Using Intel MKL on Stampede

- On login, MKL and its environment variables are loaded by default
  - They come with the Intel compiler
  - If you switch to a different compiler, you must re-load MKL explicitly (Not Recommended)

```
module swap intel gcc
module load mkl
module help mkl
```

- Compile and link for C/C++ or Fortran: dynamic linking-no Threads

```
icc   myprog.c   -mkl=sequential
ifort myprog.f90 -mkl=sequential
```

- Compile and link for C/C++ or Fortran: dynamic linking-threads

```
icc   myprog.c   -mkl=parallel
ifort myprog.f90 -mkl=parallel
```

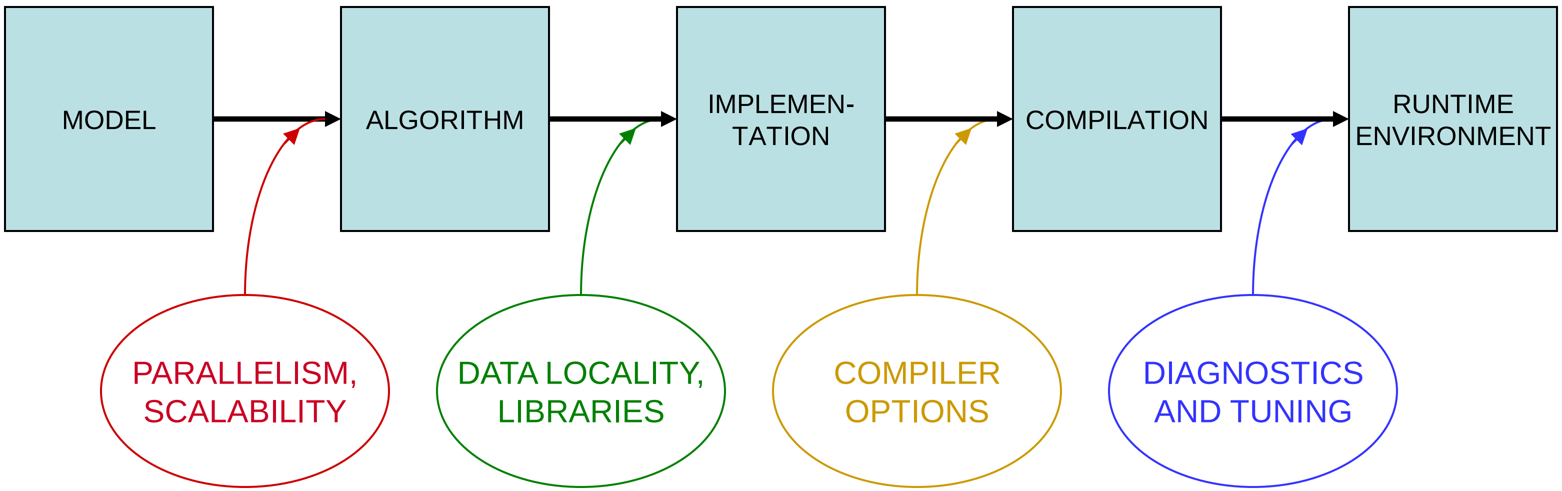
FFTW and ATLAS

- These two free libraries rely on “cache-oblivious algorithms”
  - Resulting lib is self-adapted to the hardware cache size, etc.
- FFTW, the Fastest Fourier Transform in the West
  - Cooley-Tukey with automatic performance adaptation
  - Prime Factor algorithm, best with small primes like (2, 3, 5, and 7)
  - The FFTW interface can also be linked against MKL
- ATLAS, the Automatically Tuned Linear Algebra Software
  - BLAS plus some LAPACK
  - Not pre-built for Stampede (would need to be complied from source)
  - Best to use MKL on Stampede where possible.

GSL, the GNU Scientific Library

- Complex Numbers
- Roots of Polynomials
- Special Functions
- Vectors and Matrices
- Permutations
- Sorting
- BLAS Support
- Linear Algebra
- Eigensystems
- Fast Fourier Transforms
- Quadrature
- Random Numbers
- Quasi-Random Sequences
- Random Distributions
- Statistics
- Histograms
- N-Tuples
- Monte Carlo Integration
- Simulated Annealing
- Differential Equations
- Interpolation
- Numerical Differentiation
- Chebyshev Approximation
- Series Acceleration
- Discrete Hankel Transforms
- Root-Finding
- Minimization
- Least-Squares Fitting
- Physical Constants
- IEEE Floating-Point
- Discrete Wavelet Transforms
- Basis splines

Putting Performance into Development: Compilers



Starting with how to *design* for parallelism and scalability…

…this talk is about the principles and practices during various stages of code *development* that lead to better performance on a per-core basis

Compiler Options

- There are three important categories:
  - Optimization level
  - Architecture-related options affecting performance
  - Interprocedural optimization
- Generally you will want to supply at least one option from each category

Let the Compiler Do the Optimization

- Compilers can do sophisticated optimization
  - Realize that the compiler will follow your lead
  - Structure the code so it's easy for the compiler to do the right thing (and for other humans to understand it)
  - Favor simpler language constructs (pointers and OO code won't help in hot spots)
    - Array of structs vs. structs of arrays
- Use the latest compiler and optimization options
  - Check available compiler options

```
<compiler_command> --help
```

  - The Stampede User Guide (https://portal.xsede.org/web/xup/tacc-stampede) lists compiler options affecting performance in Table 5.6
  - Experiment with combinations of options

Basic Optimization Level: -O*n*

- -O0 = no optimization: disable all optimization for fast compilation
- -O1 = compact optimization: optimize for speed, but disable optimizations which increase code size
- -O2 = default optimization
- -O3 = aggressive optimization: rearrange code more freely, e.g., perform scalar replacements, loop transformations, etc.

- Specifying -O3 is not always worth it…
  - Can make compilation more time and memory intensive
  - Might be only marginally effective
  - Carries a risk of changing code semantics and results
  - Sometimes even breaks codes!

-O2 vs. -O3

- Operations performed at default optimization level, -O2
  - Instruction rescheduling
  - Copy propagation
  - Software pipelining
  - Common sub-expression elimination
  - Prefetching
  - Some loop transformations

- Operations performed at the higher optimization level -O3
  - Aggressive prefetching
  - More loop transformations

Architecture: the Compiler Should Know the Chip

- SSE level and other capabilities depend on the exact chip
- Taking an Intel “Sandy Bridge” from Stampede as an example…
  - Supports SSE, SSE2, SSE4_1, SSE4_2, AVX
  - Supports Intel’s SSSE3 = *Supplemental* SSE3, not the same as AMD’s
  - Does *not* support AMD’s SSE5
- In Linux, a standard file shows features of your system’s architecture
  - Do this: `cat /proc/cpuinfo` {shows cpu information}
  - If you want to see even more, do a Web search on the model number
- This information can be used during compilation…

Compiler Options Affecting Performance

With Intel 13 compilers on Stampede:

- -xhost enables the highest level of vectorization supported on the processor on which you compile
- -opt-prefetch enables data prefetching
- -fast sounds pretty good, but it is not recommended
  - prevents linking with shared libraries as it implies –static
  - also implies -no-prec-div, decreasing floating point precision
- To optimize I/O on Stampede: -assume buffered_io (Fortran only)
- To optimize floating-point math: -fp=model fast[=1|2]

Interprocedural Optimization (IP)

- The Intel compilers, like most, can do IP (option -ip)
  - Limits optimizations to within individual files
  - Produces line numbers for debugging

- The Intel -ipo compiler option does more
  - Enables multi-file IP optimizations (between files)
  - Places additional information in each object file; rearranges object code
  - IP among ALL objects is performed during the load phase,
  - Can take much more time, as code is recompiled during linking
  - It is **important** to include options in **link** command (-ipo -O3 -xhost, etc.)
  - Easiest way to ensure correct linking is to link using **mpif90** or **mpicc**
  - All this works because the special Intel xild loader replaces ld
  - When archiving in a library, you must use xiar, instead of ar

Other Intel Compiler Options

- -g — generate debugging information, symbol table
- -vec_report# — {# = 0-5} turn on vector diagnostic reporting – *make sure your innermost loops are vectorized*
- -check=… -check-pointers-*=… — enable extensive runtime error checking **Should be removed for production HPC apps.**
- -openmp — multithread based on OpenMP directives
- -openmp_report# — {# = 0-2} turn on OpenMP diagnostic reporting

- Do NOT USE:
  - -static — load libs statically at runtime
  - -fast — includes -static and -no-prec-div

Best Practices for Compilers

- Recommended compiling for Stampede
  - Intel 13:
    icc/ifort -O3 -xhost -ipo prog.c/cc/f90
  - GNU 4.4 (GCC not recommended or supported):
    gcc -O3 -march=corei7-avx -mtune=corei7-avx -fwhole-program -combine prog.c
  - GNU (if absolutely necessary) mixed with icc-compiled subprograms:
    mpicc -O3 -xhost -cc=gcc -L$ICC_LIB -lirc prog.c subprog_icc.o

- -O2 is the default; compile with a different -Ox if this breaks (very rare)

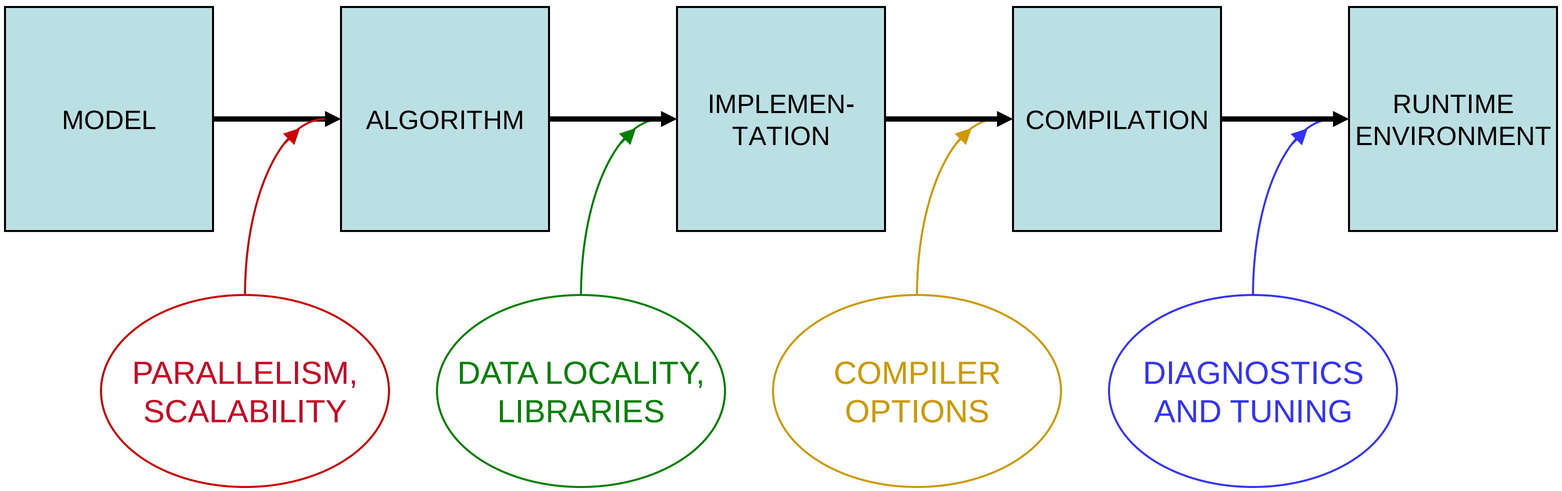
- Debug options should not be used in a production compilation
  - Compile like this only for debugging: ifort -O2 -g –check=… test.c

Lab: Compiler-Optimized Naïve Code vs. Libraries

- Challenge: how fast can we do a linear solve via LU decomposition?
- Naïve code is copied from Numerical Recipes and two alternative codes are based on calls to GSL and LAPACK
  - LAPACK references can be resolved by linking to an optimized library like ATLAS or MKL
- Compare the timings of these codes when compiled with different compilers and optimizations
  - Do ‘module load gsl’
  - Compile the codes with different flags, including “-g”, “-O2”, “-O3” (in Makefile)
  - Submit a job to see how fast the codes run (see results.txt)
  - Recompile with new flags and try again
  - Can even try to use MKL’s built-in OpenMP multithreading

- Source is in ~tg459572/LABS/ludecomp.tgz

Putting Performance into Development: Tuning



Starting with how to *design* for parallelism and scalability…

…this talk is about the principles and practices during various stages of code *development* that lead to better performance on a per-core basis

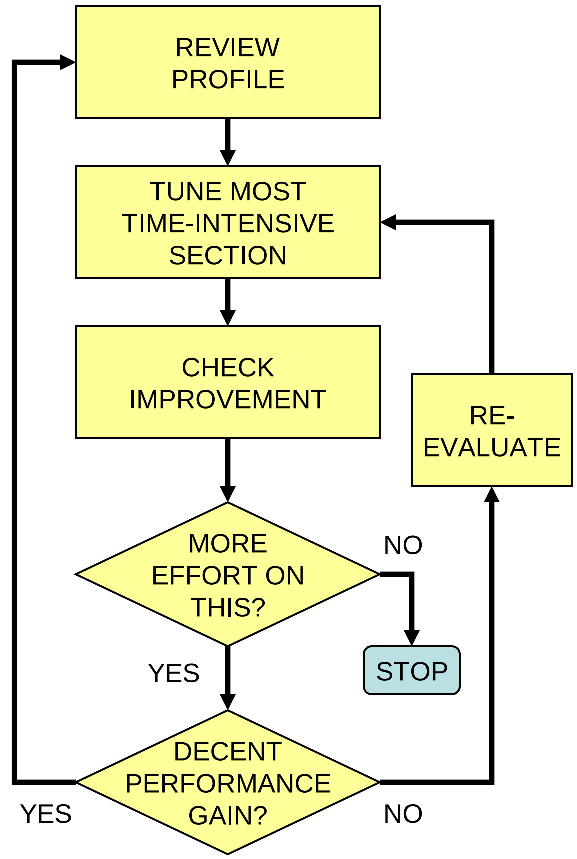
In-Depth vs. Rough Tuning

In-depth tuning is a long, iterative process:

- Profile code
- Work on most time intensive blocks
- Repeat as long as you can tolerate…

For rough tuning during development:

- Learn about common microarchitectural features (like SSE)
- Get a sense of how the compiler tries to optimize instructions, given certain features



First Rule of Thumb: Minimize Your Stride

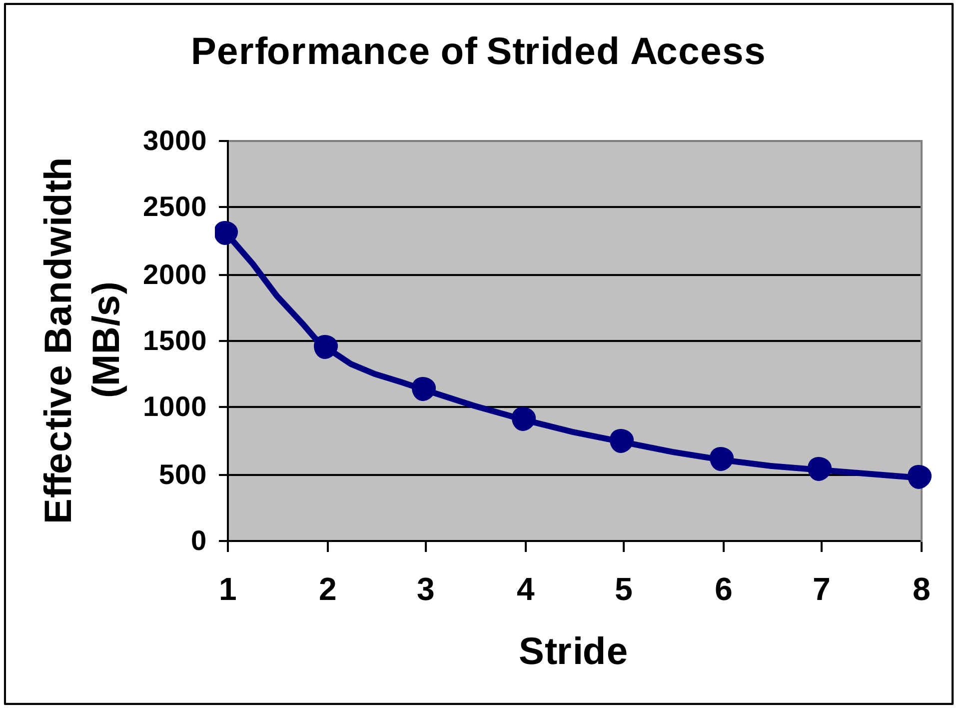
- Minimize stride length
  - It increases cache efficiency
  - It sets up hardware and software prefetching
  - Stride lengths of large powers of two are typically the worst case, leading to cache and translation look-aside buffer (TLB) misses due to limited cache associativity

- Strive for stride-1 vectorizable loops
  - Can be sent to a SIMD unit
  - Can be unrolled and pipelined
  - Can be processed by SSE and AVX instructions
  - Can be parallelized through OpenMP directives

The Penalty of Stride > 1

- 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
```



Stride 1 in Fortran and C

- The following snippets of code illustrate the correct way to access contiguous elements of a matrix, i.e., stride 1 in Fortran and C

Column Major

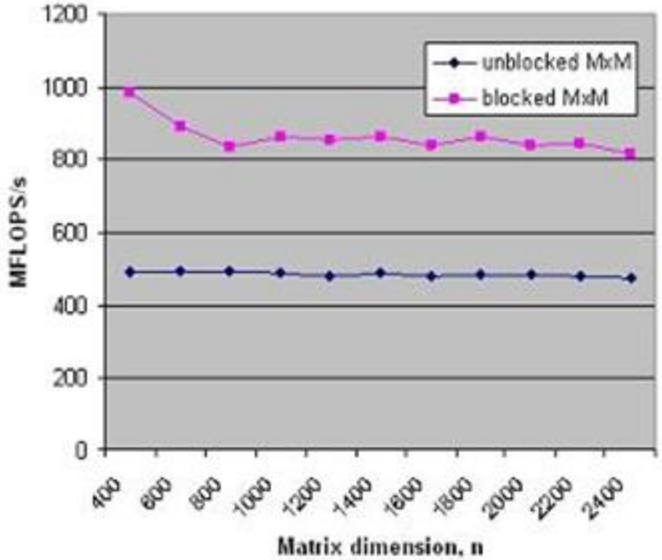
```
Fortran Example:

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
```

Row Major

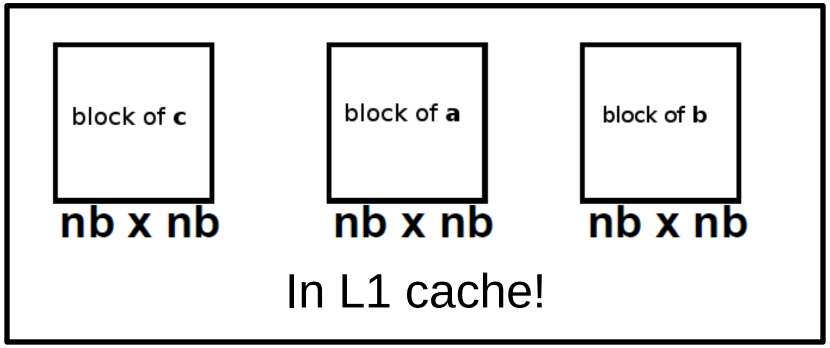
```
C Example:

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];
}
```

Loop Tiling to Fit Into Cache

Example: matrix-matrix multiplication




```
real*8 a(n,n), b(n,n), c(n,n)
do ii=1,n,nb  ! Stride by block size
  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(i,k)*b(k,j)
```



Takeaway: all the performance libraries do this, so you don't have to

Second Rule of Thumb: Inline Your Functions

- What does inlining achieve?
  - It replaces a function call with a full copy of that function’s instructions
  - It avoids putting variables on the stack, jumping, etc.

- When is inlining important?
  - When the function is a hot spot
  - When function call overhead is comparable to time spent in the routine
  - When it can benefit from Inter-Procedural Optimization

- As you develop “think inlining”
  - The C “inline” keyword provides inlining within source
  - Use -ip or -ipo to allow the compiler to inline

Example: Procedure Inlining

```
integer :: ndim=2, niter=10000000
real*8  :: x(ndim), x0(ndim), r
integer :: i, j
   ...
   do i=1,niter
      ...
      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
```

Trivial function *dist* called *niter* times

```
integer:: ndim=2, niter=10000000
real*8  :: x(ndim), x0(ndim), r
integer :: i, j
   ...
   do i=1,niter
      ...
      r=0.0
      do j=1,ndim
         r=r+(x(j)-x0(j))**2
      end do
      ...
   end do
   ...
end program
```

Low-overhead loop *j* executes *niter* times

function *dist* has been inlined inside the *i* loop

Tips for Writing Faster Code

- Write routines that can be inlined
  - Avoid calling complicated functions in hot spots.
  - Perhaps check that inlining has occurred in assembly output
- Minimize the use of pointers
- Avoid casts or type conversions, implicit or explicit
  - Conversions involve moving data between different execution units
- Avoid I/O, function calls, branches, and divisions inside loops
  - Why pay overhead over and over?
  - Move loops into the subroutine, instead of looping the subroutine call
  - Structure loops to eliminate conditionals
  - Calculate a reciprocal outside the loop and multiply inside

Best Practices from the Stampede User Guide

Additional performance can be obtained with these techniques:

- Memory subsystem tuning
  - Blocking/tiling arrays
  - Prefetching (creating multiple streams of stride-1)

- Floating-point tuning
  - Unrolling small inner loops to hide FP latencies and enable vectorization
  - Limiting use of Fortran 90+ array sections (can even compile slowly!)

- I/O tuning
  - Consolidating all I/O to and from a few large files in $SCRATCH
  - Using direct-access binary files or MPI-IO
  - Avoiding I/O to many small files, especially in one directory
  - Avoiding frequent open-and-closes (can swamp the metadata server!)

Conclusions

- Performance should be considered at every phase of application development
  - *Large-scale parallel performance* (speedup and scaling) is most influenced by choice of algorithm
  - *Per-core performance* is most influenced by the translation of the high-level API and syntax into machine code (by libraries and compilers)
- Coding style has implications for how well the code ultimately runs
- Optimization that is done for server CPUs (e.g., Intel Sandy Bridge) also serves well for accelerators and coprocessors (e.g., Intel MIC)
  - Relative speed of inter-process communication is even slower on MIC
  - MKL is optimized for MIC, too, with automatic offload of MKL calls
  - It's even more important for MIC code to vectorize well

References

- Code Optimization Virtual Workshop
  - https://www.cac.cornell.edu/VW/CodeOptimization
- Stampede User Guide:
  - https://portal.tacc.utexas.edu/user-guides/stampede