Living With Complexity: 
Pragmatic Approaches to Performance

William Gropp
wgropp.cs.Illinois.edu
With
Andreas Klöckner
Achieving High Performance is Increasingly Difficult

• Systems are increasingly complex
  • It was bad enough with caches and vector instructions, now add HBM and GPUs – and not just 1 of each
  • Multi GPU common; more than one socket/node.
• Even effective use of a single CPU core (which means using appropriate vector and other instructions) is difficult
  • Compiler vectorization requires high levels of optimization and still misses optimization opportunities (45/151 in test last week)
  • Best performance still requires specialized code, use of intrinsics, etc.
• Before we go any farther: Who is the audience for this talk?
  • People needing most/all of the available performance
  • Note that Dennard (Frequency) scaling ended ~ 2006, and since then, performance has relied on parallelism at all levels and specialization
HPC Nodes are Increasingly Complex

**DOE Sierra**
- Power 9 with 4 NVIDIA Volta GPU
- 4320 nodes
- DOE Summit similar, but
  - 6 NVIDIA GPUs/node
  - 4608 nodes

**DOE Frontier**
- AMD with 4 AMD GPU
- 100+ racks

**DOE Aurora**
- Intel SR with 6 Intel Ponte Vecchio GPUs
- Being deployed, >9K nodes

**Fugaku**
- Fujitsu A64FX (includes Vector Extensions)
- 158,976 (+) nodes

**NCSA Delta** similar but fewer racks 😊
Hardware Implications For Programs

• Heterogeneity in many ways
  • Processor – complex compute modes with scalar and vector
  • Many (but not all) include separate accelerators (GPUs and others)
  • Memory – Cache was bad enough; now HBM, other
  • I/O – Burst buffers (often violating POSIX semantics), on node, central, remote (cloud)

• For algorithm developer and programmer, the issue is Performance Heterogeneity
  • Whether the implementation uses more than one chip(let) isn’t the issue – can you see performance impact of the different elements?
  • Even vectorization counts as performance heterogeneity in this view
    • Compilers still not great at vectorizing code, and often algorithmic changes needed to take full advantage of vectorization (which specializes code, makes it hard to reason about performance)

• Impacts algorithm choice and program realization
Algorithm Considerations

- Start with the choice of mathematical model/numerical method
  - E.g., higher-order approximations for finite difference/element/volume trade floating point operations, data motion, and data size
  - Higher level choices can provide better locality
    - E.g., nonlinear Schwarz, with “local” nonlinear solves

- Performance models needed to guide algorithm design/choice
  - Model does not need to be precise – just good enough to guide
  - This is fortunate, as highly accurate performance models are very difficult to create and validate
  - But they need to be accurate enough – and many models haven’t kept up with the evolution of architectures

- One Example: Node-aware algorithms
  - Performance model captures basic system hierarchy at node level
  - Avoid redundant data copies; optimize data motion for HW characteristics
  - Suggests a different approach for process topology mapping…
MPI On Multicore Nodes

• MPI Everywhere (single core/single thread MPI processes) still common
  • Easy to think about
  • We have good performance models (or do we?)

• In reality, there are issues
  • Memory per core declining
    • Need to avoid large regions for data copies, e.g., halo cells
    • MPI implementations could share internal table, data structures
      • May only be important for extreme scale systems
  • MPI Everywhere implicitly assume uniform communication cost model
    • Limits algorithms explored, communication optimizations used

• Even here, there is much to do for
  • Algorithm designers
  • Application implementers
  • MPI implementation developers

• One example: Can we use the single core performance model for MPI?
  • \[ T = s + r n \]
  • Widely used and effective for designing parallel algorithms
  • Similar issues with logP, other models.
Rates Per MPI Process

- Ping-pong between 2 nodes using 1-16 cores on each node
- Top is BG/Q, bottom Cray XE6
- “Classic” model predicts a single curve – rates independent of the number of communicating processes
Rates Per MPI Process: 128 cores

- Increasing core count makes the situation more complex
- Note roughly similar behavior for first 32 processes
  - 1 process / core
  - 64 cores/socket
- As before, classic model predicts a single curve – rate depends only on length, independent of number of communicating processes
Why this Behavior?

• The \( T = s + r \times n \) model predicts the *same* performance independent of the number of communicating processes
  • What is going on?
  • How should we model the time for communication?
A Slightly Better Model

- For k processes sending messages, the sustained rate is
  - \( \min(R_{\text{NIC-NIC}}, k R_{\text{CORE-NIC}}) \)
- Thus
  - \( T = s + k \frac{n}{\min(R_{\text{NIC-NIC}}, k R_{\text{CORE-NIC}})} \)
- Note if \( R_{\text{NIC-NIC}} \) is very large (very fast network), this reduces to
  - \( T = s + \frac{k n}{(k R_{\text{CORE-NIC}})} = s + \frac{n}{R_{\text{CORE-NIC}}} \)
- This model is approximate; additional terms needed to capture effect of shared data paths in node, contention for shared resources, etc.
- But this new term is by far the dominant one
- This is the max-rate model (for performance limited by the maximum available bandwidth)
  - Logp model has a similar limitation and needs a similar modification
Comparison on Cray XE6

Measured Data

Max-Rate Model
Performance Model to Algorithm

- Performance measurements of halo exchange show poor communication performance
  - Bandwidth per process low relative to “ping pong” measurements
  - Easy target – blame contention in the network
- But common default mapping of processes to nodes leads to more off-node communication
  - The max rate model predicts reduced performance once $R_{\text{NIC-NIC}}$ limit reached
- We can use this to create a better, and simpler, implementation of MPI_Cart_create
Building A Better MPI_Cart_create

• Hypothesis: A better process mapping **within** a node will provide significant benefits
  • **Ignore** the internode network topology
    • Vendors have argued that their network is fast enough that process mapping isn’t necessary
    • They may be (almost) right – once data enters the network

• Idea for Cartesian Process Topologies
  • Identify nodes (see MPI_Comm_split_type)
  • Map processes **within** a node to minimize internode communication
    • Trading **intranode** for **internode** communication
    • *Using Node and Socket Information to Implement MPI Cartesian Topologies*, Parallel Computing, 2019 [https://doi.org/10.1016/j.parco.2019.01.001](https://doi.org/10.1016/j.parco.2019.01.001)
Increasing Core Count Makes Proper Mapping More Important

- Cartesian mapping on Delta
  - CPU nodes have 2 AMD Milan x 64 cores each (GPU nodes have 1 AMD Milan and 4 A100 or A40 NVIDIA GPUs)
  - Slingshot network (mostly – NIC update coming)
  - Performance in B/s (higher is better)
- Default mapping provides poor performance
  - Cart is MPI_Cart_create – also MPI_COMM_WORLD
  - Noddec uses node-awareness, inspired by max-rate model
  - Nodech extends to socket (3-level)
Is Generating Fast Executables from Modern Code a Solved Problem?

• There are some good successes – but still a challenge

• Features of successes
  • Existing languages
    • But perhaps directives/command line to fine tune semantics and choice of optimizations
  • Code transformations at various levels
  • Separate out schedule from operation (forall, iterators)

• Even transpose is tricky
  • As we’ll see in the next few slides
  • Transpose involves only data motion; no floating-point order to respect
  • Only a double loop (fewer options to consider)
A Simple Example: Dense Matrix Transpose

• do j=1,n
do i=1,n
  b(i,j) = a(j,i)
enddo
enddo

• No temporal locality (data used once)

• Spatial locality only if (words/cacheline) * n fits in cache

• Performance plummets when matrices no longer fit in cache
Blocking for Cache Helps

- do jj=1,n,stridej
  do ii=1,n,stridei
    do j=jj,min(n,jj+stridej-1)
    do i=ii,min(n,ii+stridei-1)
    b(i,j) = a(j,i)

- Good choices of stridei and stridej can improve performance by a factor of 2 or more

- But what are the choices of stridei and stridej?
  - AMD Milan, runs July 5, 2022

- For matrices too large for cache (4000x4000 for these tests), performance ranges from 2.7 to 8.1 GB/sec

- Straightforward code (-O3) provides about 3.1GB/sec
  - Best blocked code about 2.6 times as fast

- Similar results (though at lower sustained bandwidth) when running on multiple cores concurrently
  - This is the more relevant case
Why Isn’t Generating High Performance Code Really Solved?

• Assumes accurate performance model – but this is very challenging in most cases
  • Machine Learning will probably provide better ways to create/update performance models, but may be difficult to use for the second part

• Assumes manageable space of options from which to choose – but
  • Search space is huge
  • Complexity of performance behavior (even if you had an accurate model) makes it difficult to prune the search space
Code is the Enemy

• Code is a precise, executable description of an algorithm+data structure, relative to a machine model
  • Precision is good, but…
  • High-level, abstract machine models *may* make it hard to achieve performance

• How do we “solve” this (write code that gives performance) now?
  • Ignore – hope for the best from the compiler and libraries
  • Produce fast(ish) code for one system
    • Might include optimization “tricks” – loop unrolling, special vector intrinsics, vendor-specific GPU code, data structure choices (array of structures or structure of arrays or arrays of structures of arrays or …)

• A true solution must deal with challenges at all levels
  • Requires handling complexity at all levels – humans and tools typically focus on just one part of the problem
The “upstream” Problem

• In a perfect world, clever ideas get pushed into compilers/tools, and we build on them. The world is far from perfect
• Clever ideas are often also complex – hard to maintain, unexpected interactions with other parts of the code
• This argues for a combination of
  • Augmenting / extending existing languages and systems to build on existing ecosystems
  • Code transformation / writing tools to help compilers/systems
• Some of the difficult issues are in how to accomplish the combination - the “+”
Building A Code EcoSystem

- As part of two DOE-funded projects (XPACC and CEESD), we’ve been developing tools to help computational scientists focus on their science
- **Locus/ICE**
  - Manage code transformations and search among the transformations for best performance
- **Moya Just In Time Compilation**
  - Some things are only known at runtime; given that data, can produce much faster code
  - Use static analysis performed at compile time to make runtime code generation faster, better
  - “Moya-A JIT Compiler for HPC”, Programming and Performance Visualization Tools 2019
    https://link.springer.com/chapter/10.1007/978-3-030-17872-7_4
  - Note transpose results given earlier relied on compile-time choice of block size to help compiler generate good code
- **MIRGE**
  - Start at higher level representation of algorithm
  - But do so by exploiting an existing system (Python in our case), not a new language
- Of course, there are many other efforts
  - ATLAS, Spiral, FFTW, FEniCS, TCE, etc.
Practical Low-level Performance

- Processors have very complex performance behavior; extremely difficult to accurately predict performance or even order different alternatives
  - Without accurate, affordable performance model, no a priori decision can be made on which code (transformations) to use
- In practice, often need to consider alternatives
  - While compiler can do this in principle, rare and often impractical in practice
- How can you harness the power of code transformation and autotuning systems?
Locus

- Source code is annotated to define code regions
- Optimization file notation orchestrates the use of the optimization tools on the code regions defined
- Interface provides operations on the source code to invoke optimizations through:
  - Adding pragmas
  - Adding labels
  - Replacing code regions
- These operations are used by the interface to plug-in optimization tools
- Most tools are source-to-source
  - tools must understand output of previous tools
- Joint work with Thiago Teixeira and David Padua, “Managing Code Transformations for Better Performance Portability”, IJHPCA, 2019
  
  https://doi.org/10.1177%2F1094342019865606
Matrix Multiply Example

• `#pragma @LOCUS loop=matmul`
  ```
  for(i=0; i<M; i++)
    for(j=0; j<N; j++)
      for(k=0; k<K; k++)
        C[i][j] = beta*C[i][j] + alpha*A[i][k] * B[k][j];
  ```
  ```
  dim=4096;
  Search {
    buildcmd = "make clean all";
    runcmd = "/matmul";
  }
  CodeReg matmul {
    RoseLocus.Interchange(order=[0,2,1]);
    tileI = poweroftwo(2..dim);
    tileK = poweroftwo(2..dim);
    tileJ = poweroftwo(2..dim);
    Pips.Tiling(loop="0", factor=[tileI, tileK, tileJ]);
    tileI_2 = poweroftwo(2..tileI);
    tileK_2 = poweroftwo(2..tileK);
    tileJ_2 = poweroftwo(2..tileJ);
    Pips.Tiling(loop="0.0.0.0", factor=[tileI_2, tileK_2, tileJ_2]);
    tileI_3 = poweroftwo(2..tileI_2);
    tileK_3 = poweroftwo(2..tileK_2);
    tileJ_3 = poweroftwo(2..tileJ_2);
    Pips.Tiling(loop="0.0.0.0.0.0", factor=[tileI_3, tileK_3, tileJ_3]);
  } OR {
    None;
  }
  ```
Locus Generated Code
(for specific platform/size)

• #pragma @LOCUS loop=matmul
  for(i_t = 0; i_t <= 7; i_t += 1)
  for(k_t = 0; k_t <= 3; k_t += 1)
  for(j_t = 0; j_t <= 1; j_t += 1)
  for(i_t_t = 8 * i_t; i_t_t <= ((8 * i_t) + 7); i_t_t += 1)
  for(k_t_t = 256 * k_t; k_t_t <= ((256 * k_t) + 255); k_t_t += 1)
  for(j_t_t = 32 * j_t; j_t_t <= ((32 * j_t) + 31); j_t_t += 1)
  for(i = 64 * i_t_t; i <= ((64 * i_t_t) + 63); i += 1)
  for(k = 4 * k_t_t; k <= ((4 * k_t_t) + 3); k += 1)
  for(j = 64 * j_t_t; j <= ((64 * j_t_t) + 63); j += 1)
  C[i][j] = beta*C[i][j] + alpha*A[i][k]*B[k][j];
DGEMM by Matrix Size

DGEMM on IBM Power

- XLC base
- locus+XLC 2lt
- GCC base
- locus+GCC 2lt
- locus+GCC 3lt

DGEMM on Intel x86

- ICC base
- Locus+ICC 2lt
- Locus+ICC 3lt
- Intel MKL

Matrices shape (n*n)
Tuning Must be in a Representative Environment

- For most processors and regular (e.g., vectorizable) computations
  - Memory bandwidth for a *chip* is much larger than needed by a single *core*
  - *Share of* memory bandwidth for a *core* (with all cores accessing memory) is much smaller than needed to avoid waiting on memory

- Performance tests on a single core can be very misleading
  - Example follows
  - Can use simple MPI tools to explore dependence on using one to all cores
    - See baseenv package
  - Ask this question when you review papers 😊
Stencil Sweeps

• Common operation for PDE solvers
  • Structured are often “matrix free”
  • Unstructured and structured mesh stencils have low ”computational intensity” — number of floating-point operations per bytes moved

• Conventional wisdom is that cache blocking and similar optimizations are ineffective
  • For example, “Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors” argues this, and provides experimental data to support it

• But the analysis and experiments are usually based on one core per chip/socket
  • And the number of cores has grown substantially since 2007
  • What if every core is executing a stencil sweep?
void heat3d(double A[2][N+2][N+2][N+2]) {
int i, j, t, k;
#pragma @LOCUS loop=heat3d
for(t = 0; t < T-1; t++) {
for(i = 1; i < N+1; i++) {
for(j = 1; j < N+1; j++) {
for (k = 1; k < N+1; k++) {
A[(t+1)%2][i][j][k] = 0.125 * (A[t%2][i+1][j][k] - 2.0 * A[t%2][i][j][k] + A[t%2][i-1][j][k]) + 0.125 * (A[t%2][i][j+1][k] - 2.0 * A[t%2][i][j][k] + A[t%2][i][j-1][k]) + 0.125 * (A[t%2][i][j][k-1] - 2.0 * A[t%2][i][j][k] + A[t%2][i][j][k+1]) + A[t%2][i][j][k];
}
}
}
}
A High Level Approach

• Start with Python
  • High level language with strong software ecosystem
  • Integrate with code transformation/generation tools to create high-performance versions

• Alternative to creating a new Domain Specific Language

• Center for Exascale-Enabled Scramjet Design
  • Ceesd.Illinois.edu
  • Coupled hypersonic fluid flow with combustion and material interaction
  • Target is DOE Exascale systems – nodes with multiple accelerators
  • Changing nodes – IBM P9+NVIDIA to AMD+AMD (and Intel+Intel if ANL included)
MIRGE Overview

**CONTROL Layer (Python)**

while \( t < t_{\text{final}} \):

- \( p = \text{eos}(u) \)
- \( d = \text{diff}(u, p) \)
- \( f = \text{flux}(u, p) \)
- \( b = \text{bcs}(u, p) \)
- \( s = \text{surf}(u, p) \)
- \( r = \text{rhs}(d, f, b, s) \)
- \( u_{\text{next}} = \text{adv}(u, r) \)

**Kernels**

- \( p = \text{fct}(u, e) \)
- \( T = a_0 + a_1 u + a_2 u^2 + \ldots \)
- \( \sum_i D^{\text{diff}}_{u_k} u_{ki} \) \( \text{DG diff} \)
- \( v - \frac{\partial}{\partial x} U \) \( \text{DG flux} \)
- Material Models: Black Box (WARP, Puma, SPARTA)
- Volume + Flux + BCs
- Time Advance: \( u + \Delta t(u, u, \ldots) \)

**Intermediate Representation (DAG of kernels)**

- Fusion
- DAG of macrokernels

**Execution / Computation**

- Code Generation
- Cache
- PTX
- C
- SPIR-V

**Transform Engines**

- Adapter
- ROSE
- Adapter
- PIPS
- Adapter
- Loopy

**Machine Expert Interface**

- Metadata + Machine X: transform recipe A
- Metadata + Machine Y: transform recipe B

**Machine Expert Interface**

- Device + Scheduler
- Device + Scheduler
- Device + Scheduler

**NCSA**
Early Performance Results

- Abstractions visible to app:
  - numpy-like array, nested containers thereof
  - Array op. indirection layer (use Jax, Pytato, Numpy, eager GPU)
  - Metadata ("tags") describe arrays, axes in app. Terms
- Pipeline of intermediate representations
  - Array DFG ("pytato") via lazy eval, lowered to
  - Imperative, polyhedral ("loopy") represetnation, lowered to
  - OpenCL (for execution)
- Transformations (currently)
  - On Array DFG: Metadata prop., materialization, redundant exprs.
  - On loop IR: Loop/kernel fusion, array contraction, tile and prefetch
  - Driven by app-aware transform code using metadata
- Organizational unit for tile/prefetch: “Fused einsum”
- Numerical method is DG-FEM
- Performance measured on single Nvidia Titan V GPU
- Work of Kaushik Kulkarni and Andreas Klöckner
Summary and Challenges

- Achieving performance is hard
  - Compilers, Libraries, and tools can help
  - But complexity of real systems requires tuning, which implies flexibility in code generation
  - Relatively simple performance models can help answer “Is this as fast as it should be?”

- Leverage existing systems: “build on the shoulders of giants”

- Build on software ecosystem to realize algorithms
  - Need to consider high and low level needs – and address separately but compatibly

- Need to embrace composition of programming systems, address “+”