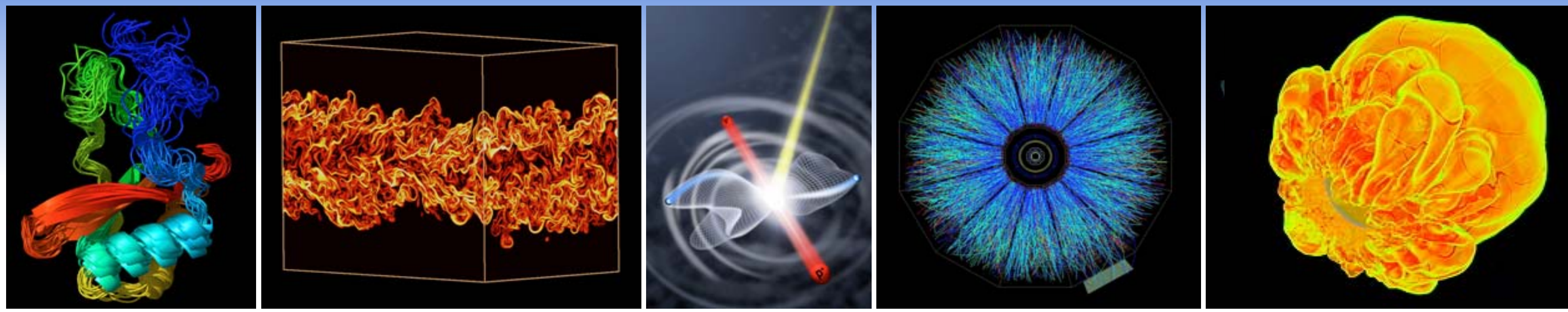


Autotuning: The *Big* Questions

John Shalf
Lawrence Berkeley National Laboratory

August 10, 2009

Times They are a Changing
(trends in computer architecture)

- **Clock Frequency Scaling Has Ended**
  - Now we double cores every 18 months instead of doubling clock frequency
  - Alternate trajectory is manycore (GPUs, etc.): start with hundreds of simpler cores
  - Future “speed up” requires strong scaling from explicit parallelism
- **Memory capacity per computational element will be decreasing**
  - Also forces us towards strong scaling, *even if you don’t want it*
  - Requires constant memory footprint in face of exponential scaling
- **Memory and communication bandwidth per peak FLOP decreasing**
  - Old optimization target was to reduce flops (increase communication)
  - New optimization target is to reduce communication (increase FLOPs)
- **Architectural diversity is increasing *(architectural uncertainty)***
  - Current languages are mis-matched with emerging machine models
  - Performance portability is more of a problem than ever
- **Load imbalance is increasingly problematic with larger parallelism**
- **Reliability for largest-scale systems is likely going down**

Role of Auto-Tuning
(hiding architectural complexity)

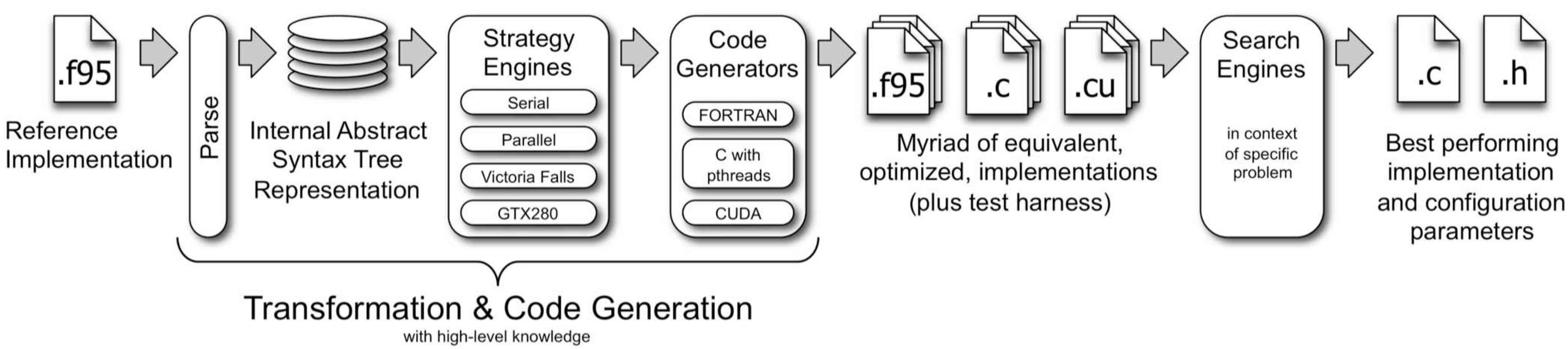
- **Present higher level of abstraction to hide architectural diversity**
  - Abstraction of algorithm implementation is a shim for poorly understood (*or broken*) abstract machine model
- **Automate search through optimization space to achieve performance portability and strong scaling**
  - Some focus on search through optimization parameters
  - More aggressive schemes (with higher level abstractions) search through alternate strategies (e.g. super-solvers)
- **Automate insertion of memory movement directives (prefetch or DMA) to economize on memory bandwidth**
- **Provides abstractions that decouple "cores" from data decomposition**
  - Currently abstract data layout
  - *Perhaps can also abstract heterogeneous code (functional partitioning of algorithms)*
- **Provides abstractions that enable easier hierarchical parallelism**
- ***Could search through alternative balancing strategies?***
- ***Could code generation hide reliability and fault detection methods into algorithms?***
- ***Could code generation hide energy/performance trade-offs?***

Challenges for Existing Auto-Tuning Infrastructure

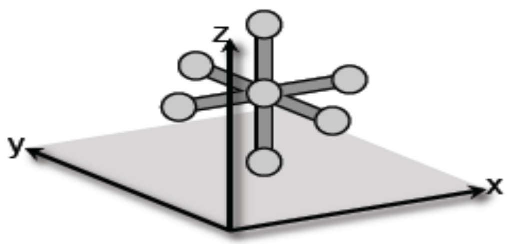
- **Coverage**
  - Can we cover enough 'motifs' using domain-specific frameworks approach?
  - Can we offer a sufficient level of abstraction with a loop-oriented "autotuning compiler" approach?
- **Parallelization & communication strategy**
  - Current auto-tuning primarily focuses on scalar opt
  - How will we incorporate more variation on parallel strategy?
- **Search**
  - Minimizing search space (search pruning)
  - Optimizing search strategy (machine learning, dynamic programming).
- **Improving Interface to users (integrating with apps)**
  - Creating interfaces for library design experts (rapid re-tuning of libraries)
  - Creating domain-optimized interfaces (F77 as a DSL)
  - Integrating with existing frameworks (Cactus, Chombo)
  - SEJITS (just-in-time specialization of runtime compiled scripting languages)

Generalized Stencil Auto-tuning Framework

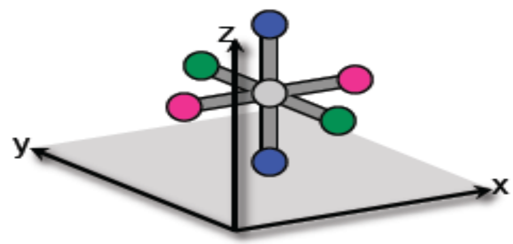
- **Ability to tune many stencil-like kernels**
  - No need to write kernel-specific perl scripts
  - Uses semantic information from existing Fortran
- **Target multiple architectures**
  - Search over many optimizations for each architecture
  - Currently supports multi/manycore, GPUs
- **Better performance = Better energy efficiency**



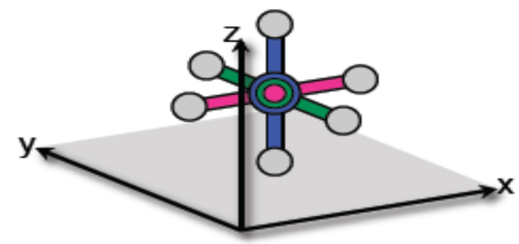
Multi-Targeted Auto-Tuning



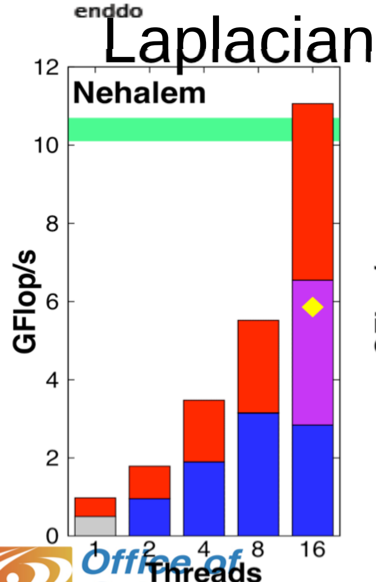

```
do k=2,nz-1,1
do j=2,ny-1,1
do i=2,nx-1,1

 uNext(i,j,k)=
  alpha*u(i,j,k)+
  beta*(u(i+1,j,k)+u(i-1,j,k)+
        u(i,j+1,k)+u(i,j-1,k)+
        u(i,j,k+1)+u(i,j,k-1)
       )
enddo
enddo
enddo
```

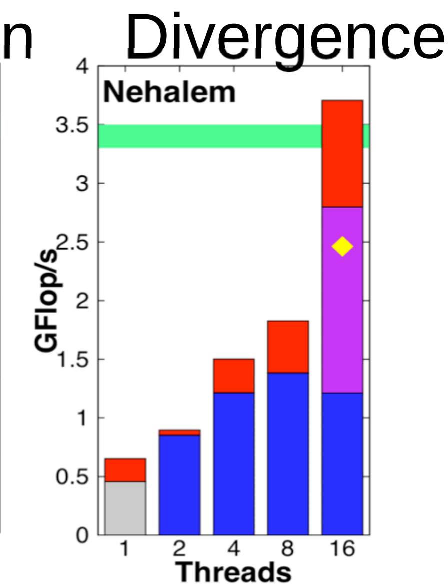


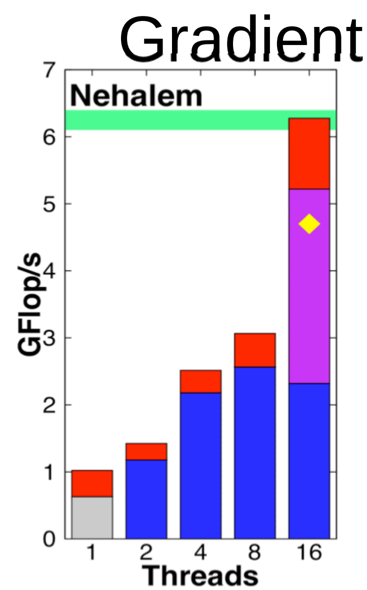

```
do k=2,nz-1,1
do j=2,ny-1,1
do i=2,nx-1,1

 u(i,j,k)=
  alpha*( x(i+1,j,k)-x(i-1,j,k) )+
   beta*( y(i,j+1,k)-y(i,j-1,k) )+
  gamma*( z(i,j,k+1)-z(i,j,k-1) )

enddo
enddo
enddo
```

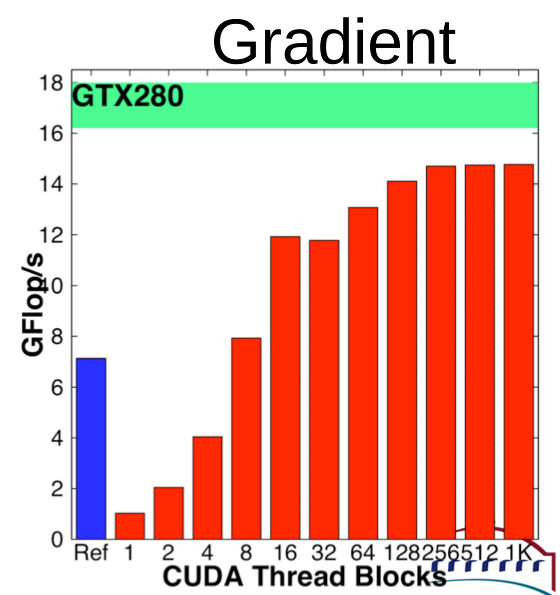



```
do k=2,nz-1,1
do j=2,ny-1,1
do i=2,nx-1,1

 x(i,j,k)=alpha*( u(i+1,j,k)-u(i-1,j,k) )
 y(i,j,k)= beta*( u(i,j+1,k)-u(i,j-1,k) )
 z(i,j,k)=gamma*( u(i,j,k+1)-u(i,j,k-1) )

enddo
enddo
enddo
```



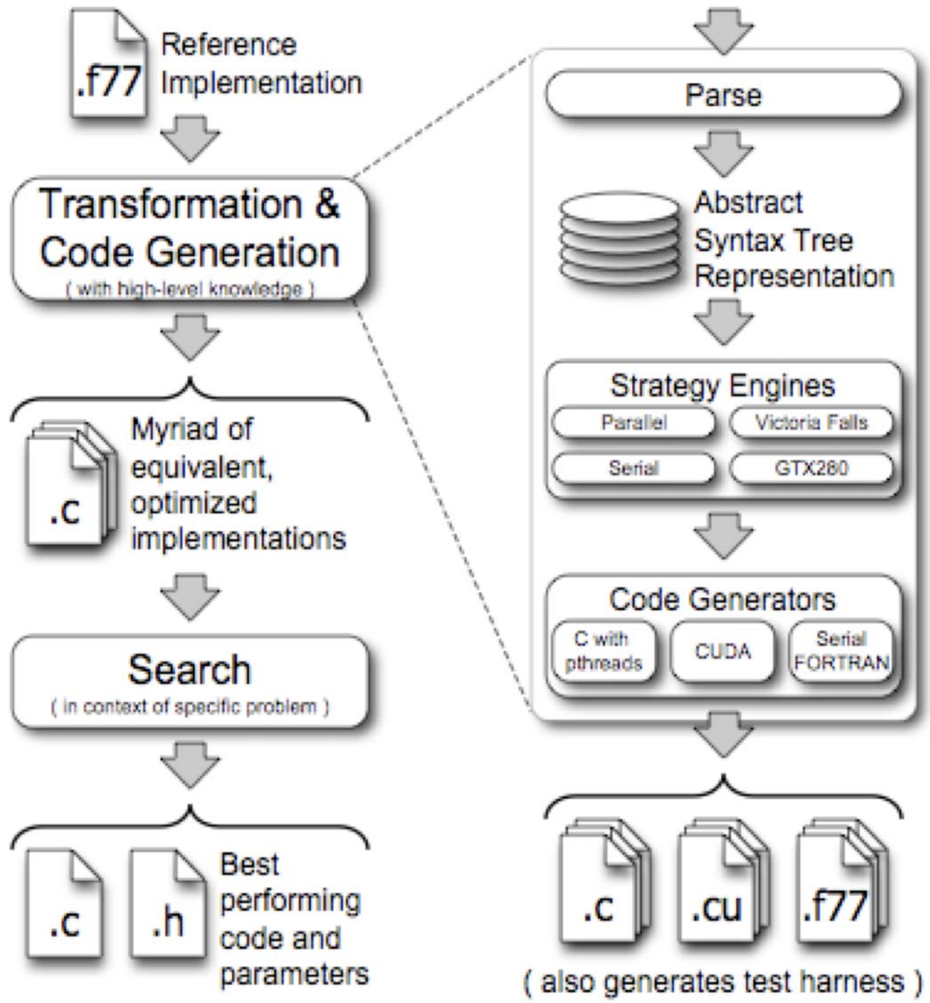






Framework for Stencil Auto-Tuning
(F77 as domain-specific language)

- **Framework can make code maintenance easier**
  - **Annotated kernels**
  - **Integrated verification suites (Cactus)**
- **Next: Wizard for reducing work and maintaining kernels specs**
- **Integrate with existing Cactus/ Chombo frameworks**
- **Enables analysis of inter-kernel dataflow to do**
  - **schedule for communication hiding,**
  - **local store reuse**
  - **functional partitioning**



Automatic Search for Multigrid

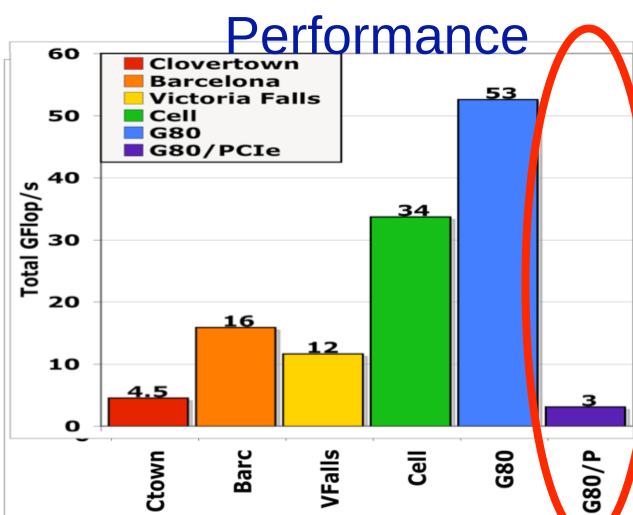
(Cy Chan / Shoaib Kamil)

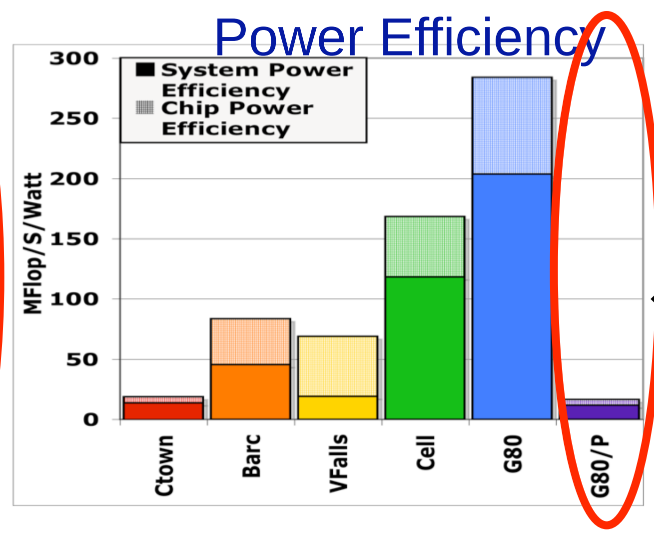
- **Combines auto-tuning with strategy optimization**
  - Stencil auto-tuner for prolongation, restriction, relaxation operators
  - Measure convergence rate for V-cycle vs. bottom solve (estimate at every level of V-cycle)
  - Measure performance of prolongation, restriction, repartitioning, relaxation operators
- **Uses dynamic programming to select optimal combination of V-cycle, repartion, and bottom solve**
- **Keyed off of problem-specific numerical algorithm behavior (not just cycle rate)**
- **Where else can auto-tuner observe convergence behavior to auto-select runtime strategy? (supersolvers + autotuners)**

Scheduling For Heterogeneity

(move towards global search for optimal schedule)

- **Most autotuning focuses on standalone kernel performance**
- **Heterogeneous systems with non-uniform memory access need focus on data movement optimization**
  - **This is likely a global (cross-kernel) optimization problem**
  - **Combinatorial explosion of options (is it tractable even with search optimizations?)**





When G80 communicates with host through PCIe, performance benefits are greatly reduced

Functional Partitioning

(different parallelization strategy for strong scaling without domain decomposition)

- **Need abstraction to decouple notion of “thread” from problem domain decomposition**
- **When strong scaling, you eventually run out of ability to further decompose domains (all ghost-cells in stencil case)**
  - Then what do you do?
- **Functional partitioning**
  - Have multigrid solver running concurrently with physics in climate code
  - Or have subset of cores handle communication or load balancing
- **For every machine, you need a different problem partitioning**
  - Auto-tuners can hide the partitioning strategy (automate search through different partitioning conformations)
  - Need code specification that explicitly identifies concurrent heterogeneous “functionality” to run concurrently (or use dataflow analysis)
  - Load imbalance is not bad if the tail on the imbalance is bounded
- **Partitioning search can be extended to heterogeneous hardware (not just heterogeneous partitioning of code)**
- **Dataflow scheduling is NOT a problem for humans to solve!**

Search for Optimal Communication Strategy

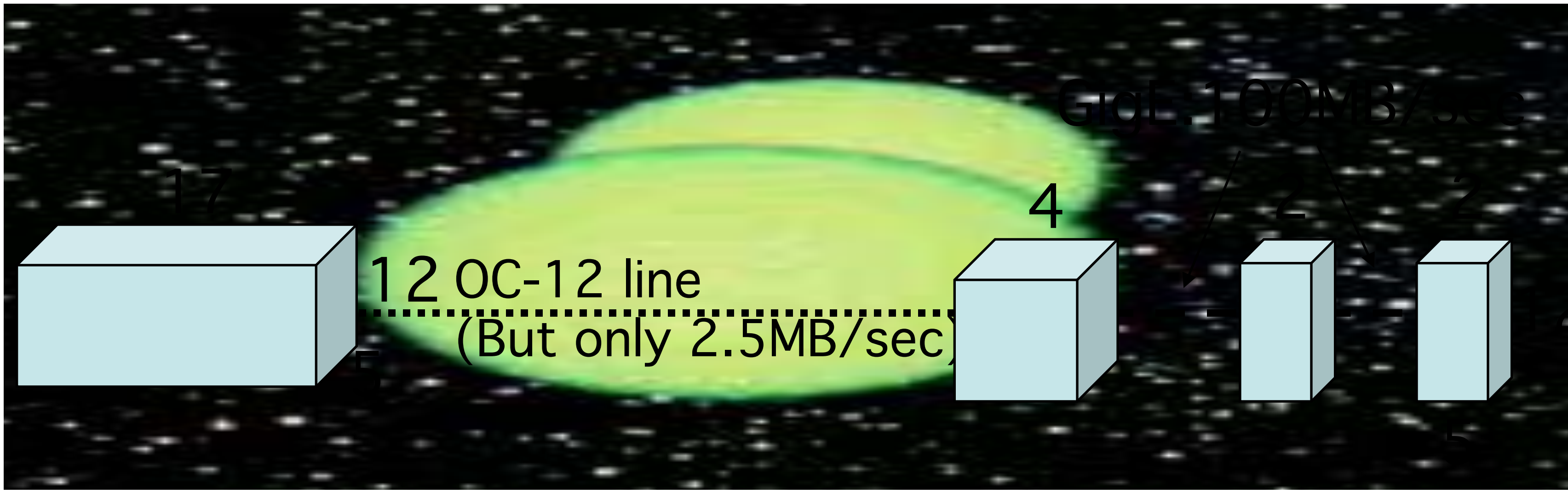
- **Most auto-tuning focuses on serial optimization (parallel optimization is separate step)**
- **Examples of parallel optimization of collectives (Rajesh), but still distinct from serial optimizations**
  – runtime adaptive tuning (rather than offline tuning)
- **Optimize inter-processor communication strategy (requires more substantial algorithm reorganization to identify legal strategies)**
  – Not sure if compiler based auto-tuners can do this or not (challenge)

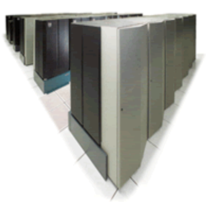
Runtime Adaptive Search for Communication Strategy

- **Most auto-tuning is founded on offline performance analysis**

- **Consider runtime adaptive auto-tuning rather than offline search**

Runtime Adaptive Distributed Computation (with Argonne/U.Chicago)





SDSC IBM SP
1024 procs
5x12x17 =1020

NCSA Origin Array
256+128+128
5x12x(4+2+2) =480

This experiment:

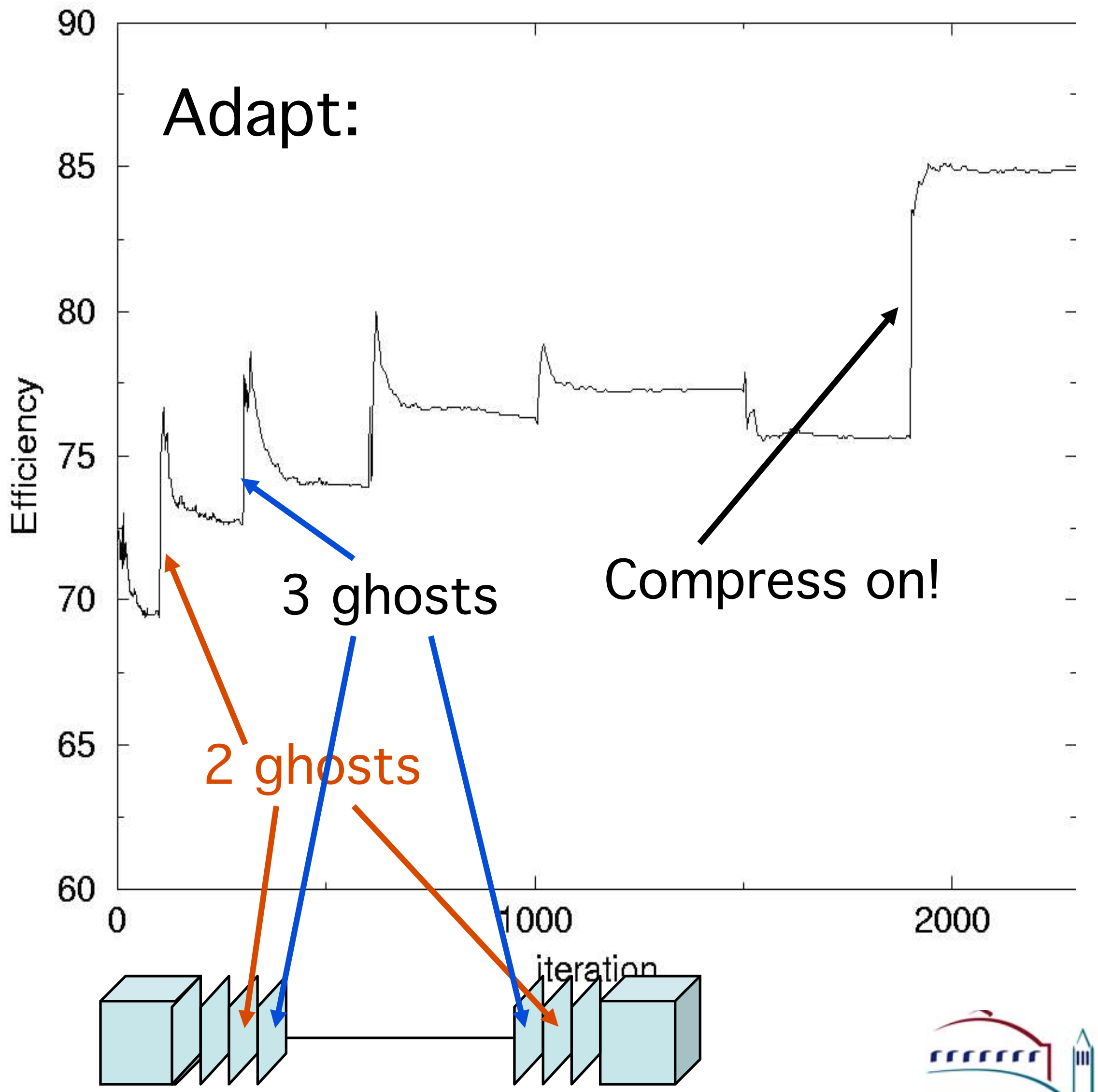
- Einstein Equations (but could be any Cactus application)

Achieved:

- First runs: 15% scaling
- With new techniques: 70-85% scaling, ~ 250GF

Dynamic Runtime Adaptation

- Automatically adapt to bandwidth latency issues
- Application has NO KNOWLEDGE of machines(s) it is on, networks, etc
- Adaptive techniques make NO assumptions about network
- Adaptive MPI unigrid driver required NO changes to the physics components of the application!! (plug-n-play!)
- Issues:
  - More intellegent adaption algorithm
  - Eg if network conditions change faster than adaption…



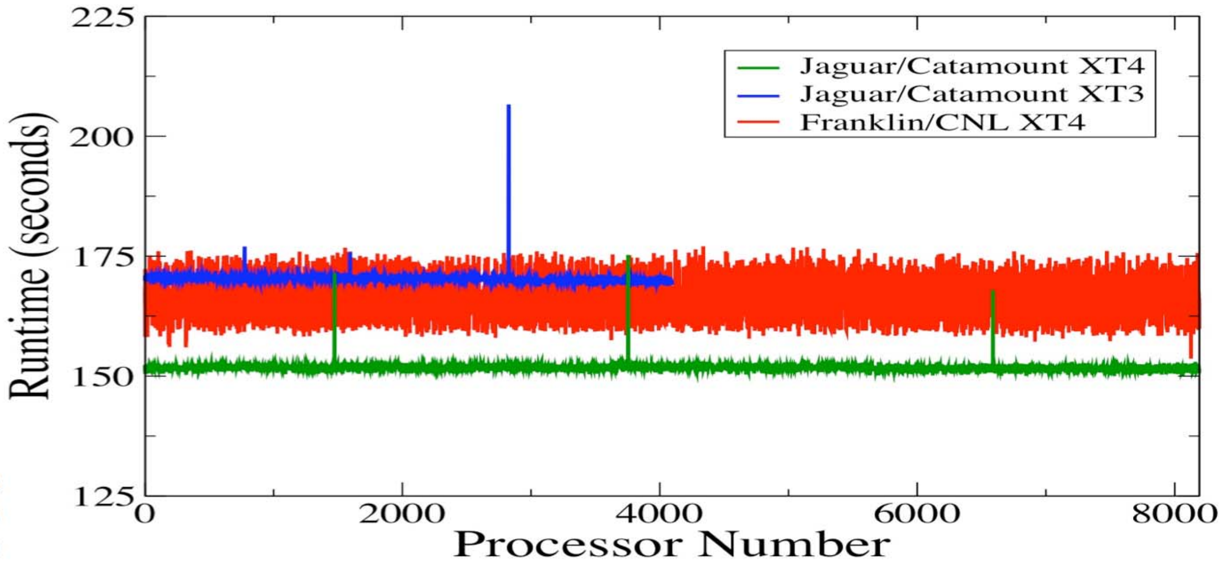
Fault Resilience and Load Balance

- There are many strategies for load balancing
- Difficult problem for users to solve
  - Want pervasive instrumentation for fault resilience
  - But resulting code is messy and tedious
- Perhaps auto-tuners can play a role to insert hooks for state migration and hide machine-specific load-balancing strategies
  - depends on communication characteristics of system
- *Extension of search for optimal problem partitioning for heterogeneous architectures*

Load Imbalances and Resilience

(is managing load-balance a subset of runtime autotuning?)

- **Adaptive Algorithms result in load imbalances**
- **Fine grained power management & hard fault mgmt. makes even homogeneous resources look heterogeneous**
- **Fault resilience introduces inhomogeneity in execution rates *(error correction is not instantaneous)***
- **Most load balancers are build on poorly understood heuristics**
  - ***Can we automate the search for optimal load-balancing strategy?***
  - ***Can we use auto-tuning to hide fault tedious resilience instrumentation?***



Uncertainty Quantification and Extended Precision

- **Automate insertion of software extended precision arithmetic for UQ**

- **Automate insertion of software extended precision, or optimize all-reduce collectives (runtime tuning)**

Now the Negative Part of This Presentation

- **Which problem are we trying to solve?**
  - Autotuning is becoming a heavily overloaded term (and we are rapidly layering on additional requirements)
  - Require more disambiguation to move forward productively

- **Our Machine Model is Fundamentally Broken**
  - Is auto-tuning the right way to hide this, or are more fundamental changes required

Autotuning Disambiguation

- **Which problem are we trying to solve?**
  - Automate tuning libraries for expert library designers?
  - Create simpler/convenient interfaces for novice scientists?
  - Are we trying to create higher-level abstraction for broken machine model?
  - *Solution target points to radically different approaches*

Segmenting Developer Roles
(and not calling it ALL auto-tuning)

Developer Roles	Domain Expertise	CS/Coding Expertise	Hardware Expertise
Application: *Assemble solver modules to solve science problems. (eg. combine hydro+GR +elliptic solver w/MPI driver for Neutron Star simulation)*	Einstein	Elvis	Mort
Solver: *Write solver modules to implement algorithms. Solvers use driver layer to implement "idiom for parallelism". (e.g. an elliptic solver or hydrodynamics solver)*	Elvis	Einstein	Elvis
Driver: *Write low-level data allocation/placement, communication and scheduling to implement "idiom for parallelism" for a given "dwarf". (e.g. PUGH)*	Mort	Elvis	Einstein

Strategies

- **Automating process of library tuning for minor architectural variants**
  - The compiler approach with loop annotations is a great approach
  - Bad for domain scientists (who don't even know what the params mean), but great for experts
  - Works fine if machine model is just a preturbation of norm
- **Making convenient interface for domain scientists**
  - Novices should *not* be exposed to hardware-based tuning
  - Even writing "loop nests" is against productivity (go to higher-level abstractions)
  - Provide "wizard" interfaces to reduce keystrokes to specify solution
  - Limited coverage, but can we cover enough?
  - Can we create framework to make it faster to create such application-specific wizards
- **Hiding radical machine model differences**
  - Fortran or C are too imperative (overly specify solution)
  - Have to infer "intent" of code or do a *lot* of work to expose constraints to enable legal transformations
  - Perhaps C/Fortran are wrong level of abstraction to enable the required transformations (bigger arch differences force us to higher-level of abstraction to achieve unity)

Broken Machine Model

Broken Machine Model
(is "explicit search" the right approach?)

- **Our Machine Model is Fundamentally Broken**
  - Is auto-tuning the right way to hide this, or are more fundamental advances required
  - Are compiler-based auto-tuners operating at wrong level of abstraction to hide fundamental differences in machine model?
- **Critique of auto-tuning on serial machines: hides the fact that we no longer understand what HW is doing**
- **Is "explicit search" the correct alternative to fixing a fundamentally broken machine model (and commensurate fixes to our programming model?)**

Evidence Machine Model is Broken (memory)

- **Machine model doesn't reflect characteristics of emerging machines**
  - PRAM model (presumes equal communication costs)
  - But on-chip communication is 100x lower latency and 10x higher bandwidth than off-chip!
  - Ignoring these differences results in huge inefficiencies!
- **Evidence: Cache-dependent programming model obfuscates memory locality**
  - Cache virtualizes main memory addresses
  - But difference in cost of data transfer between on-chip vs. off-chip memory is HUGE
  - Wrong to pretend they are the same (and that is what cache forces us to do)
  - Local-store explicitly differentiates between on-chip and off chip memory addresses, but no abstraction to program it

Evidence of Broken Machine Model

- **Are emerging machine models even commensurable?**
- **OpenCL does not *(and fundamentally cannot)* target performance portability**
  - Its not even on the design targets
  - Current evidence suggests that lack of performance portability will not be fixed by more mature code-generation back end (requires more fundamental re-write of kernels)
  - Means that OpenCL is good as output target for auto-tuners, but inappropriate level of abstraction for input target for directive guided compiler-based auto-tuners
- **Can we as a community have more than one programming model (result of incommensurable abstract machine models)?**
- **Can auto-tuning forestall this undesirable outcome?**

More Evidence of Broken Machine Model

- Domain Decomposition is the primary approach to parallel speed-up
  - Formula is relatively well understood
- Feed-forward pipelines are not very easy to express
  - Unbounded side-effects make this complicated
- If we think functional-partitioning and feed-forward pipelines are important, then there is something wrong with a pmodel that makes it hard to express

Using Functional/Dataflow Approach

- **Make cost of data movement first-class citizen**
- **Requires understanding of scope of side-effects**
  - **Make code that is more analyzable (functional programming)**
  - **Use strong typed constructs to make analysis easier for runtime (Ct single-assignment arrays/TStructs)**
  - **Annotate code to identify data (IVY)**
- **Goal: know what memory is touched by unit of code**
  - **Runtime as dataflow work scheduler**
  - **auto-tuning search to optimize logistics of data movement**
- **Side-benefit: easier to identify minimal state to preserve for checkpoint/rollback**
  - **If you know what data is modified when, then can do fine-grained recovery of state if unit of execution fails**
- ***Use as foundation for autotuning infrastructure***

Source of Load Imbalances

(is managing load-balance a subset of runtime autotuning?)

- **Fine grained power management makes even homogeneous cores look heterogeneous**
- **Nonuniformities in process technology creates non-uniform operating characteristics for cores on a CMP**
- **To improve chip yield, disable cores with hard errors *(impacts locality of chip-level interconnects)***
- **Fault resilience introduces inhomogeneity in execution rates *(error correction is not instantaneous)***

Source of Load Imbalances

(is managing load-balance a subset of runtime autotuning?)

- **Fine grained power management makes even homogeneous cores look heterogeneous**
- **Nonuniformities in process technology creates non-uniform operating characteristics for cores on a CMP**
- **To improve chip yield, disable cores with hard errors (impacts locality of chip-level interconnects)**
- **Fault resilience introduces inhomogeneity in execution rates (error correction is not instantaneous)**

Heterogeneity is going to be pervasive problem for programmers even if not intentional design!