“Industrial” Auto-tuning using CrayATF

Adrian Tate
Technical Lead of Scientific Libraries
Senior Software Engineer, Cray Inc.
iWAPT, Tokyo, Oct 2009
History of Cray Supercomputers

- Cray-1 (1976)
- Cray-XMP (1982)
- Cray-2 (1985)
- Cray-YMP (1988)
- Cray-C90 (1991)
- Cray-T3D (1993)
- Cray-T90 (1994)
- Cray-T3E (1995)
- Cray-SV1 (2001)
- Cray-XT3 (2005)
- Cray-XT5 (2008)
<table>
<thead>
<tr>
<th>Year</th>
<th>System</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1976</td>
<td>Cray-1</td>
<td>Single Vector Pipe, No data cache, One–few processors</td>
</tr>
<tr>
<td>1982</td>
<td>Cray-XMP</td>
<td>Multiple Pipe, small data cache, Several processors</td>
</tr>
<tr>
<td>1985</td>
<td>Cray-2</td>
<td>Massively parallel, Data caches, Distributed memory</td>
</tr>
<tr>
<td>1988</td>
<td>Cray-YMP</td>
<td>Massively parallel, vector, scalar, x86, CISC, GPU, FPGA, multi-core</td>
</tr>
<tr>
<td>1991</td>
<td>Cray-C90</td>
<td></td>
</tr>
<tr>
<td>1993</td>
<td>Cray-T90</td>
<td></td>
</tr>
<tr>
<td>1994</td>
<td>Cray-T3D</td>
<td></td>
</tr>
<tr>
<td>1995</td>
<td>Cray-T3E</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>Cray-SV1</td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Cray-X1</td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>Cray-XT3</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>Cray-XT5</td>
<td></td>
</tr>
</tbody>
</table>
### Hardware Trends

<table>
<thead>
<tr>
<th>Year</th>
<th>BLAS1</th>
<th>LINPACK</th>
<th>BLAS2</th>
<th>BLAS3</th>
<th>LAPACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1976</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1982</td>
<td></td>
<td></td>
<td>Single Vector Pipe</td>
<td>No data cache</td>
<td>One–few processors</td>
</tr>
<tr>
<td>1985</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1988</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1991</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1993</td>
<td></td>
<td></td>
<td>Multiple Pipe</td>
<td>small data cache</td>
<td>Several processors</td>
</tr>
</tbody>
</table>

- **Cray-1**
- **Cray-XMP**
- **Cray-2**
- **Cray-YMP**
- **Cray-C90**
- **Cray-T90**

**Software Tools**

- ScaLAPACK
- PETSc
- ATLAS
- FFTW
- Trilinos

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Cray-T3D**
- **Cray-T3E**
- **Cray-SV1**
- **Cray-X1**
- **Cray-XT3**
- **Cray-XT5**

**Hardware Trends**

- Massively parallel
- Data caches
- Distributed memory
- Massively parallel, vector, scalar, x86, CISC, GPU, FPGA, multi-core
What is the role of vendor in libraries tuning today?

- Clearly, not to make the problem worse
- Improve performance of PETSc and Trilinos on Cray MPPs
  - tuning sparse matrix vector multiply in general fashion
- Tune HPL benchmark for largest machines (massive runtime)
  - $O(N^3)$ factorization driven by multiple parameters
- Tune Dense linear algebra (BLAS3 mainly)
  - BLAS3
- Apply the above only to the Cray hardware
  - Allows the search space to be manipulated to our advantage
- Tune eigensolvers in a general purpose way

- It is pretty obvious that hand-tuning alone cannot achieve this
Can we construct a generalized AT framework to do all the above?
Look at 3 examples: HPL

- HPL (High Performance Linpack)
  - $O(N^3)$ factorization and solve
  - Parameter tuning is now paramount
  - Has 13 parameters (+ 7 more in Cray version)
  - Some parameters have very large dimensionality
  - Search space is very large indeed (more later)
  - Has become a massive problem due to excessive runtime
Desired flow for HPL Auto-tuning

1. Form problem configuration
2. XML input configuration
3. Construct Test Program
4. Compile code versions
5. Define next search
6. Deduce concurrency in search
7. Construct input files
8. Construct Batch interface
9. Spawn threads, one for each job
10. Create input file
11. Execute HPL instance
12. Spin on completion
13. Search completed?
14. Optimal HPL configuration
Desired flow for HPL Auto-tuning

1. Form problem configuration
2. XML input configuration
3. Construct Test Program
4. Compile code versions
5. Define next search
6. Deduce concurrency in search
7. Construct input files
8. Construct Batch interface
9. Spawn threads, one for each job
10. Create input file
11. Execute HPL instance
12. Spin on completion
13. Search completed?

Optimal HPL configuration
Look at 3 examples: Sparse Linear Algebra

- Sparse Linear Algebra (mainly sparse matrix-vector product)
  - (for CSR) Irregular memory access
  - Memory bandwidth bound kernel
  - Wildly dependent on matrix characteristics
    - Has never had a general purpose tuned code for this reason
Desired Sparse Linear Algebra flow

1. Kernel Template
2. Parser: read template
3. Deduce all combinations of transformation
4. Generate Code Variant
5. Search complete?
6. Compile testing program
7. Deduce concurrency in search
8. Perform Search
9. Search complete?
10. Performance model
11. Optimized kernel execution
12. User code calls High Level Library
13. Matrix categorized
14. Best kernel for matrix category chosen
Desired Sparse Linear Algebra flow

Offline

Search limits → Matrix Dimensions → Sparsity/Density

XML input specification

Kernel Template → Parser: read template → Deduce all combinations of transformation

Generate Code Variant

Kernel Variant → Search complete?

Compile testing program

Deduce concurrency in search

Perform Search

Search complete?

Performance model

User code calls High Level Library

Matrix categorized

Best kernel found: apply cache! Use!

Optimized kernel execution

Runtime
Example 3: Dense Linear Algebra

- Mostly serial $O(N^3)$ BLAS3 optimizations
- Loop transformations
- Multiple algorithmic effects
Desired Dense Linear Algebra Flow

- Algorithm A Template
- Algorithm B Template
- Algorithm N Template

- Parse Template
- Parse Template
- Parse Template

- Deduce all combinations of transformation
- Generate code variant

- Search complete?
- Search complete?

- Compile testing program
- Concurrent in search
- Perform Search
- Search complete?

- Performance model
- Best kernel for matrix category chosen

- User code calls tuned Library
- Optimized kernel execution
Desired Dense Linear Algebra Flow
“Industrial” Auto-tuning Model

- Search space is made “manageable” because of
  - Restriction to one processor type
  - Knowledge of target problem sizes / characteristics
- Search space is attainable because of infinite resource
- Freedom only to make incremental changes (e.g. no new data-structures)
- Hence, to make an auto-tuner that works in the real world
  - Enormous Offline Testing infrastructure
    - We have unlimited resources available for the offline testing!
  - Performance model as output from offline autotuning
    - We can assume the same architecture for each distribution!
  - Adaptive libraries that take the performance model as input
- The above define our “industrial” autotuning model
- CrayATF is the framework built on this model
Overlay and group the functionality just described

<table>
<thead>
<tr>
<th>Code Generator</th>
<th>Search Engine</th>
<th>Execution Engine</th>
<th>Input Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse Template file</td>
<td>Deduce concurrency in search</td>
<td>Construct batch interface</td>
<td>Provide generic XML input interface</td>
</tr>
<tr>
<td>Translate directives to code transforms</td>
<td>Create new search table</td>
<td>Take information from Search engine</td>
<td>Input matrix characteristic</td>
</tr>
<tr>
<td>Deduce # transformations</td>
<td>Check search completion</td>
<td>Spawn threads,</td>
<td>Input problem sizes</td>
</tr>
<tr>
<td>Produc[e specific kernel variant</td>
<td>Create Performance Model</td>
<td>Create input files</td>
<td>Input searching limits</td>
</tr>
<tr>
<td>Parse multiple algorithm templates</td>
<td>Create Performance Model</td>
<td>Execute codes in parallel</td>
<td>Enter matrix characteristics</td>
</tr>
</tbody>
</table>
HPL as components

- **Code Generator**
  - Parse Template file
  - Translate directives to code transforms
  - Deduce # transformations
  - Produce specific kernel variant
  - Parse multiple algorithm templates

- **Search Engine**
  - Deduce concurrency in search
  - Create new search table
  - Check search completion
  - Create Performance Model

- **Execution Engine**
  - Construct Batch interface
  - Take information from Search engine
  - Spawn threads
  - Create input files
  - Execute codes in parallel
  - Spin on completion

- **Input Module**
  - Provide generic XML input interface
  - Input matrix characteristic
  - Input problem sizes
  - Input searching limits
  - Enter matrix characteristics
Sparse Linear Algebra as components

Code Generator
- Parse Template file
- Translate directives to code transforms
- Deduce # transformations
- Produce specific kernel variant
- Parse multiple algorithm templates

Search Engine
- Deduce concurrency in search
- Create new search table
- Check search completion
- Create Performance Model

Execution Engine
- Construct batch interface
- Take information from Search engine
- Spawn threads, create input files
- Create input files
- Execute codes in parallel
- Spin on completion

Input Module
- Provide generic XML input interface
- Input matrix characteristic
- Input problem sizes
- Input searching limits
- Enter matrix characteristics
Dense Linear Algebra as components

- **Code Generator**
  - Parse Template file
  - Translate directives to code transforms
  - Deduce # transformations
  - Produce specific kernel variant
  - Parse multiple algorithm templates

- **Search Engine**
  - Deduce concurrency in search
  - Create new search table
  - Check search completion
  - Create Performance Model

- **Execution Engine**
  - Construct batch interface
  - Take information from Search engine
  - Spawn threads
  - Create input files
  - Execute codes in parallel
  - Spin on completion

- **Input Module**
  - Provide generic XML input interface
  - Input matrix characteristic
  - Input problem sizes
  - Input searching limits
  - Enter matrix characteristics
CrayATF – a Generic and Modular Framework

Most importantly – this is
a) extensible
b) replaceable
Initial sets of parameters (random/user specified)

Parameter specifications 
(range, step size, dependency, priority)

Execute a single iteration of the search algorithm

Need more tuning?

Generate sets of parameters in the next execution phase

DONE!

Search Module

Code Generator/Build Module
(Ruby)

Batch Module
(Ruby)

Generate Kernels

Build program

Execute all program executions

Get performance numbers

Input
(XML)

Search Module
(Ruby)
Input XML Files

Machine specifications (directories, PBS options, max_cores, walltime)

Search Module (Ruby)

Search table. Each row is a unique list of param values to be executed.

Batch Module (Ruby)

For each row in search table:

Launch multiple ruby threads in parallel

Create unique input file, Create PBS script

Launch job

Wait for job end

Parse output file

Append execution data to search table

Thread barrier

Create unique input file, Create PBS script

Launch job

Wait for job end

Parse output file

Append execution data to search table

Receive results from Batch module

More tuning ??

Yes

No

DONE!
Why Ruby?

- Ruby is the language used for almost all ATF development
- Scripting ability
  - E.g. One-line text replacement of a single file

```ruby
subs.keys.each { |x| filestring.gsub!(x, subs[x]) }
```

- System programming ease:
  - E.g. On Cray XT systems, find all the jobs I have in the queue, and delete them:

```ruby
out = Array.new(`qstat -u #{$whoami.to_s}.to_s.to_a`)
5.upto(out.length-1){|line| 
  system("qdel #{out[line].split('.')[0].to_i}")
}
```
• Extremely simple and lightweight threading
  • Threadpool implemented in 40 lines of code includes routines to:
    • Initialize the pool
    • Launch threads
    • Destroy threads
    • Exception handling

• Super-soft typed
  • For non-numerical work, we do not want to be concerned with
    • Datatype conversion
    • Accuracy
    • Performance (!)
  • Allows functional code to be developed very quickly

• Integration with XML for extremely powerful configuration/input methods
High Performance Linpack benchmark
- Used for top500 rankings
- Traditional tuning approach for HPL:
  1. **Choose N to fill local memory** (reduce comms cost)
  2. **heavily tune serial dgemm (parallel dgemm dominates)**
  3. **find a good enough parameter combination** (trial and error)

- This has been successful in the past, but
- #1 is hard to do when the machine grows so large
- #3 has never been taken very seriously in practice
- But does have good auto-tuning treatment - Hollingsworth et al
Then Cray delivered a small machine to ORNL
Then Cray delivered a small machine to ORNL

200 cabinets of XT5 (HE)
18,772 nodes of 8 core nodes
37,544 sockets of AMD Barcelona
300 TB of main memory

Traditional method:
matrix dimension of $N = 6,122,903$
This equates to a HPL runtime of:

39 hours

MTTI of brand new system – a few hours

Probability of completing a 39 hour job = 0.00%

JaguarPF was given to Cray ATF team
HPL parameter tuning

- 17 HPL parameters + Cray’s additional parameters + Programming model options
- An example of sensitivity of a single parameter:

<table>
<thead>
<tr>
<th>NB</th>
<th>bcast</th>
<th>pmap</th>
<th>pfact</th>
<th>nbmin</th>
<th>ndiv</th>
<th>rfact</th>
<th>depth</th>
<th>swap</th>
<th>thresh</th>
<th>transL</th>
<th>transU</th>
<th>EQUIL</th>
<th>align</th>
<th>P</th>
<th>N</th>
<th>time</th>
<th>%peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>140</td>
<td>3429286</td>
<td>483.38</td>
<td>51.70%</td>
</tr>
<tr>
<td>160</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>140</td>
<td>3429286</td>
<td>313.47</td>
<td>79.71%</td>
</tr>
</tbody>
</table>

- Exhaustive search space is measured in **tens of years** runtime
- Typically, studies reduce the search by reducing scale
- However, early progress of ATF-HPL showed that:
  - **parameter information from small scale does not translate to full scale**
“Grouped and Attributed Orthogonal Search”

- We can’t consider most search algorithms
  - Within only 5% of optimal would be a disaster for top500 list
- Use Grouped, Attributed, Orthogonal search
1. Define list of parameters to be studied

\[ p_0 \quad p_1 \quad p_m \]
2) Define Groupings between parameters

$p_0 \quad p_1 \quad \cdots \quad p_m$
3) Define attributes for each group based on attributes for each parameter

Attributes for this group
- Requires full scale
- Requires small memory
- Can tolerate early completion
- Needs to be varied wildly

\( p_0 \ p_1 \ p_m \)
4) Loop over each group
4) Loop over each group
5) Expand length of each parameter
5) Perform Search within group
(holding all other parameters steady)
5) Take the best performing result & carry the best parameter values to the next search
5) Define next search group
(keeping best from last search)
In practice

- GOAS gets very close to optimal
- At expense of large search space
- At expense of huge amount of man-power

- Knowledge of hardware and algorithm allows very sensible selection of groups
  - Reduces the search space by knowledge

- Although it is not elegant, GAOS cannot be beaten in our tests
## ATF in use - Nov 2008 top500 list

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>Vendor</th>
<th>Cores</th>
<th>RMax</th>
<th>RPeak</th>
<th>Nmax</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DOE/NNSA/LANL</td>
<td>IBM</td>
<td>129600</td>
<td>110500</td>
<td>1456700</td>
<td>2329599</td>
<td>2483.47</td>
</tr>
<tr>
<td>2</td>
<td>Oak Ridge National Laboratory</td>
<td>Cray Inc.</td>
<td>150152</td>
<td>105900</td>
<td>1381400</td>
<td>4712799</td>
<td>6950.6</td>
</tr>
<tr>
<td>3</td>
<td>NASA/Ames Research Center/NAS</td>
<td>SGI</td>
<td>51200</td>
<td>487005</td>
<td>608829</td>
<td>2300760</td>
<td>2090</td>
</tr>
<tr>
<td>4</td>
<td>DOE/NNSA/LLNL</td>
<td>IBM</td>
<td>212992</td>
<td>478200</td>
<td>596378</td>
<td>2456063</td>
<td>2329.6</td>
</tr>
<tr>
<td>5</td>
<td>Argonne National Laboratory</td>
<td>IBM</td>
<td>163840</td>
<td>450300</td>
<td>557056</td>
<td>2580479</td>
<td>1260</td>
</tr>
<tr>
<td>6</td>
<td>Texas Advanced Computing Center</td>
<td>Sun</td>
<td>62976</td>
<td>433200</td>
<td>579379</td>
<td>0</td>
<td>2000</td>
</tr>
<tr>
<td>7</td>
<td>NERSC/LBNL</td>
<td>Cray Inc.</td>
<td>38642</td>
<td>266300</td>
<td>355506</td>
<td>1612399</td>
<td>1150</td>
</tr>
<tr>
<td>8</td>
<td>Oak Ridge National Laboratory</td>
<td>Cray Inc.</td>
<td>30976</td>
<td>205000</td>
<td>260200</td>
<td>2466816</td>
<td>1580.71</td>
</tr>
<tr>
<td>9</td>
<td>NNSA/Sandia National Laboratories</td>
<td>Cray Inc.</td>
<td>38208</td>
<td>204200</td>
<td>284000</td>
<td>2500000</td>
<td>2506</td>
</tr>
<tr>
<td>10</td>
<td>Shanghai Supercomputer Center</td>
<td>Dawning</td>
<td>30720</td>
<td>180600</td>
<td>233472</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Cray Adaptive Sparse Kernel (CASK)

- Cray Adaptive Sparse Kernels – The Crown Jewel of CrayATF
- The CASK Process
  1. Offline – produce all code variants for tuning strategy
  2. Offline – define target matrix classifications
  3. Offline – produce performance model for given matrix class
  4. Runtime – analyze matrix and deduce classification
  5. Runtime – assign tuned kernel to user code

- CASK silently sits beneath PETSc on Cray systems
  - Trilinos support coming soon
- CASK ATF flow looks very like the flow shown earlier
- **CASK released with PETSc 3.0 in February 2009**
  - Generic and blocked CSR format
CASK and PETSc: Single Node XT5

Speed-up of PETSc + CASK versus PETSc
Speedup on Parallel SpMV on 8 cores
60 different matrix classifications
CASK Scalability (XT4)

SpMV performance only

Performance of PETSc + CASK VS PETSc
N=65,536 to 67,108,864

Full solver with incomplete Cholesky local preconditioning

Performance of PETSc + CASK vs. PETSc
N=65,536 to 67,108,864

GFlops vs. # of Cores

- MatMult-CASK
- MatMult-PETSc

GFlops vs. # of Cores

- BlockJacobi-IC(0)-CASK
- BlockJacobi-IC(0)-PETSc
Lessons

- When you build an infrastructure for “industrial” purposes
- Search spaces should be manipulated via your knowledge of hardware
- At least 50% of the effort is pure software engineering
- Languages like Ruby and Python make things realistic
- Should not get too attached to what is “auto-tuning”
  - Whatever works for our problems is what we need to do
  - We do not care about definitions
  - Search algorithms are only interesting if they help us achieve our goals
  - It seems that there are emerging several distinct sub-classes of auto-tuning
- We found new uses for auto-tuning in the process:
  - Sanity/stability testing of new hardware
  - Excellent regression test for libraries
ATF is not a generalized auto-tuner for scientific applications
  - It is practical design tailored for vendor tuning of libraries

We did not make auto-tuning easy
  - In our case, it required one of the best teams in the industry to be 100% devoted for many many months

CrayATF is in its infancy, not nearing completion
Cray will provide hybrid next generation XT system with GPUs in 2010
Cray will provide a HPC Programming Environment for hybrid system

On this system, the tuning approach is **HIGHLY** parameterized
  - Which algorithm is best?
  - Number of blocks per matrix?
  - How much matrix to GPU/CPU?
  - How to schedule transfer to GPU?
  - Number of threads per block?
  - Shape of threads per block?
  - What type of memory to use?

ATF for GPUs will be main approach into library GPU tuning
Thankyou

- Q&A