Bench-testing Environment for Automated Software Tuning (BEAST)

Programming Autotuners

Presenter: Piotr Luszczek
Collaborators: Jakub Kurzak, Mark Gates, Hartwig Anzt
Students: Yaohung (Mike) Tsai, Blake Haugen
Motivation

\[ c_{ij} = \sum_k a_{ik} b_{kj} \]

\[ S_{abij} = \sum_{ck} \left( \sum_{df} \left( \sum_{el} B_{befl} \times D_{cde} \right) \times C_{dfjk} \right) \times A_{acik} \]

\[ \forall B_i = \cdot POTRF(A_i) \]
\[ \forall B_i = \cdot GEQRF(A_i) \]
\[ \forall B_i = \cdot GETRF(A_i) \]

\[ O_{nkprq} = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} F_{k,c,r,s} D_{n,c,g(p,u,R,r,h)} \cdots \]
Compilation vs. Autotuning

- **Compilation**
  - Uses very limited autotuning
  - Works for all codes
  - Finishes in seconds
  - Obeys the language syntax
  - Optimizes for machine model
  - Performs better for fixed sizes

- **Autotuning**
  - Often relies on the compiler
  - Works for some codes
  - Finishes when optimized
  - Delivers correct math
  - Optimizes over experimental data
  - Specializes in fixed sizes
Example: $C = AB$

$$c_{ij} = \sum_k a_{ik} b_{kj}$$
Example: C = AB - Parameters

- dim_m
- dim_n
- blk_m
- blk_n
- blk_k
- blk_m_a
- blk_n_a
- blk_m_b
- blk_n_b
- Vectorization
- Use shmem
- ...

- 15 parameters
- Exponential search space
- Many parameter combinations are invalid due to limitations in
  - Hardware
  - Software
  - Algorithm
Problem with Manual Iteration

For dim_m = 32:1024
  - For dim_n = 32:1024
    • For blk_m = dim_m:dim_m:maxM
      - For blk_n = dim_n:dim_n:maxN
        • For blk_k = 16:maxK
          • For vectorize = “yes”, “no”
            • For fetch_A = “yes”, “no”
              • For texture = “none”, “1D”, “2D”
                • ...

But make sure that:
  - dim_m*dim_n doesn't exceed the number of thread blocks for the tested card
  - There is enough shared memory
  - There is enough registers
  - Maintain occupancy levels above threshold
  - ...

Constraints
Constraints' Basics

- Constraints allow the code generator to substantially prune the search space
- There are three categories of constraints
  - **Hard**
    - Based on hardware specification
      - Total threads
      - Maximum threads per block
  - **Soft**
    - Based on expected performance
      - Occupancy level
      - #FMAs per LOAD
  - **Correctness**
    - Based on algorithmic formulation
      - Divisibility of sizes by blocking factors
      - Numerical correctness
Iterator Basics: Declarative Approach

- **Expression iterators**
  - `dim_m = range( 32, max_threads_dim_x, 32 )`
  - `blk_m = range( dim_m, maxM, dim_m )`

- **Function iterators**
  - `@beast.iterator`
    ```python
    def blk_n_a():
        x = blk_k
        if trans_a != 0:
            x = blk_m
        return range(x, 0, -1)
    ```

- **Closure iterators**
  - `@beast.iterator`
    ```python
    def fibonacci():
        prev = next = 1
        while next <= largest_number:
            yield next
            next, prev = next+prev, next
    ```
Condition Basics

- Conditions express any of the three kinds of constraints
- Conditions as expressions
  - `over_max_threads = beast.condition(block_threads > max_threads_per_block)`
- Closure conditions
  - `@beast.condition
def over_max_shmem():
    return block_shmem > max_shared_mem_per_block`
Performance of Autotuning in Python

Higher is better.

<table>
<thead>
<tr>
<th></th>
<th>1 loop</th>
<th>2 loops</th>
<th>3 loops</th>
<th>4 loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python while</td>
<td>4.5</td>
<td>4.5</td>
<td>4.6</td>
<td>4.4</td>
</tr>
<tr>
<td>Python range</td>
<td>4.8</td>
<td>6.3</td>
<td>6.5</td>
<td>6.1</td>
</tr>
<tr>
<td>Python xrange</td>
<td>6.6</td>
<td>6.7</td>
<td>6.8</td>
<td>6.4</td>
</tr>
</tbody>
</table>
Performance of Autotuning in Lua

Higher is better.

<table>
<thead>
<tr>
<th></th>
<th>1 loop</th>
<th>2 loops</th>
<th>3 loops</th>
<th>4 loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lua while</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lua repeat</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lua for</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance of Autotuning in C, Java, Fortran

Higher is better.

- C: 1 loop: 948, 2 loops: 1296, 3 loops: 1294, 4 loops: 1294
- Java: 1 loop: 925, 2 loops: 1269, 3 loops: 1297, 4 loops: 1297
- Fortran: 1 loop: 997, 2 loops: 1295, 3 loops: 1292, 4 loops: 1292

Million iterations per second
Optimizations Details

- The code generator figures out the optimal order of loop nests
- Iterators become loops with proper nesting
- The nesting is determined by the dependence DAG
- Conditions have to be checked as early as possible to prune the search space
  - Compiler equivalent: loop invariant code motion
- Type inference keeps the generated code fast
  - Scripting language iteration may be orders of magnitude slower

<table>
<thead>
<tr>
<th>Language</th>
<th>Performance (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>7</td>
</tr>
<tr>
<td>Lua</td>
<td>32</td>
</tr>
<tr>
<td>C</td>
<td>1296</td>
</tr>
<tr>
<td>Java</td>
<td>1270</td>
</tr>
<tr>
<td>Fortran</td>
<td>1295</td>
</tr>
</tbody>
</table>
Optimizations: Example

- dim_n
- dim_m
- vector
- blk_k
- blk_n
- blk_m_a
- blk_n_a
- blk_m_b
- blk_n_b

- low occupancy shmem

- enough shmem
- enough threads
- sufficient occupancy
- dimensions are congruent
for (dim_n = 32; dim_n < 1025; dim_n += 32)
    for (vector = 0; vector < 2; vector += 1)
        for (dim_m = 32; dim_m < 1025; dim_m += 32)
            for (blk_k = 16; blk_k < 64; blk_k += 16)
                for (blk_n = dim_n; blk_n < maxN + 1; blk_n += dim_n)
                    for (blk_m = dim_m; blk_m < maxM + 1; blk_m += dim_m) {
                        blk_m_a_type_len = 1;
                        if (vector != 0)
                            blk_m_a_type_len = dim_vec;
                        blk_m_a_x = floor(blk_m / blk_m_a_type_len);
                        if (trans_a != 0)
                            blk_m_a_x = floor(blk_k / blk_m_a_type_len);
                        for (blk_m_a = blk_m_a_x; blk_m_a < 0; blk_m_a += -blk_m_a_type_len) {
                            blk_n_a_x = blk_k;
                            if (trans_a != 0)
                                blk_n_a_x = blk_m;
                            for (blk_n_a = blk_n_a_x; blk_n_a < 0; blk_n_a += -1) {
                                blk_n_b_x = blk_n;
                                if (trans_b != 0)
                                    blk_n_b_x = blk_k;
                                for (blk_n_b = blk_n_b_x; blk_n_b < 0; blk_n_b += -1) {
                                    blk_m_b_type_len = 1;
                                    if (vector != 0)
                                        blk_m_b_type_len = dim_vec;
                                    blk_m_b_x = floor(blk_k / blk_m_b_type_len);
                                    if (trans_b != 0)
                                        blk_m_b_x = floor(blk_n / blk_m_b_type_len);
                                    for (blk_m_b = blk_m_b_x; blk_m_b < 0; blk_m_b += -blk_m_b_type_len)
BEAST Design

NVIDIA Kepler
Intel Xeon Phi
AMD Fiji

extract hardware information

kernel stencil

verification

generation & pruning engine

extract compilation information

Distributed benchmarking engine

Extract performance information

workstation
server
MPP

Machine learning engine

principal components
decision trees
genetic algorithms

Adjust tuning parameters

BEAST Design
Bench-testing Environment for Automated Software Tuning

http://icl.utk.edu/beast/

analysis and reporting
profiles
charts
projections

http://icl.utk.edu/beast/
Performance: the Traditional View

![Graph showing performance metrics vs. kernel ID](image-url)
Data Analysis: Convex Hull
Hierarchical Clustering of GPU Metrics
Future Work

• Apply autotuning to new kernels
• Continue work on parallel code compilation and autotuning
  – Multilevel parallelism: OpenMP and MPI
• Add new language features to the code generators
• Integration of the generated code with existing libraries
The Road Ahead