PetaBricks: Use of Autoturning to bring Robust and Portable Performance (and Parallelism) to the Average Programmer

Saman Amarasinghe

with

Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Outline

• Target Audience
• Why We Need a New Language
• PetaBricks Language
• PetaBricks Compiler
• PetaBricks Performance
• Variable Accuracy
Today: The Happily Oblivious Average Joe Programmer

• Joe is oblivious about the processor
  – Moore’s law bring Joe performance
  – Sufficient for Joe’s requirements

• Joe has built a solid boundary between Hardware and Software
  – High level languages abstract away the processors
    – Ex: Java bytecode is machine independent

• This abstraction has provided a lot of freedom for Joe

• Parallel Programming is only practiced by a few experts
Squandering of the Moore’s Dividend

• 10,000x performance gain in 30 years! (~46% per year)
• Where did this performance go?

• Last decade we concentrated on correctness and programmer productivity
• Little to no emphasis on performance
• This is reflected in:
  – Languages
  – Tools
  – Research
  – Education
Matrix Multiply
An Example of Unchecked Excesses

<table>
<thead>
<tr>
<th>Immutable</th>
<th>ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17,094,152</td>
</tr>
</tbody>
</table>

- Abstraction and Software Engineering
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transposing
  - Tiling
- Vectorization
- Prefetching
- Parallelization
Matrix Multiply
An Example of Unchecked Excesses

• Typical Software Engineering Approach
  – In Java
  – Object oriented
  – Immutable
  – Abstract types
  – No memory optimizations
  – No parallelization

296,260x

• Good Performance Engineering Approach
  – In C/Assembly
  – Memory optimized (blocked)
  – BLAS libraries
  – Parallelized (to 4 cores)

• In Comparison: Lowest to Highest MPG in transportation

294,000x
Joe the Parallel Programmer

- Moore’s law is not bringing anymore performance gains
- If Joe needs performance he has to deal with multicores
  - Joe has to deal with performance
  - Joe has to deal with parallelism

"C’mon, Joey. If you wanna see a Multicore Performance, you gotta put up with a little Parallel Programming."
Can Joe Handle Both Parallelism and Performance Tuning?

**Current Trajectory**

Programmer handles parallelism and performance tuning.

**Better Trajectory**


*Today*

Programmer is oblivious to performance.
Outline

• Target Audience
• Why We Need a New Language
  • PetaBricks Language
  • PetaBricks Compiler
  • PetaBricks Performance
  • Variable Accuracy
Ancient Days...

- Computers had limited power
- Compiling was a daunting task
- Languages helped by limiting choice
- Overconstraint programming languages that express only a single choice of:
  - Algorithm
  - Iteration order
  - Data layout
  - Parallelism strategy
...as we progressed....

- Computers got faster
- More cycles available to the compiler
- Wanted to optimize the programs, to make them run better and faster
...and we ended up at

- Computers are extremely powerful
- Compilers want to do a lot
- But…the same old overconstraint languages
  - They don’t provide too many choices
- Heroic analysis to rediscover some of the choices
  - Data dependence analysis
  - Data flow analysis
  - Alias analysis
  - Shape analysis
  - Interprocedural analysis
  - Loop analysis
  - Parallelization analysis
  - Information flow analysis
  - Escape analysis
  - …
PetaBricks: Rethinking Languages

- **Algorithmic Choice**
  - Programmer: Provides multiple algorithms
  - Compiler: Finds the best hybrid algorithm

- **Iteration Order**
  - Programmer: Provides producer-consumer relationships
  - Compiler: Synthesizes the outer loops for the best parallel and sequential execution orders

- **Variable Precision**
  - Programmer: Describes the required precision
  - Compiler: Finds the fastest converging hybrid algorithm and termination condition(s)
Outline

• Target Audience
• Why We Need a New Language
• PetaBricks Language
• PetaBricks Compiler
• PetaBricks Performance
• Variable Accuracy
transform MatrixMultiply
from A[c,h], B[w,c]
to AB[w,h]
{
    // Base case, compute a single element
    to(AB.cell(x,y) out)
    from(A.row(y) a, B.column(x) b) {
        out = dot(a, b);
    }
}
PetaBricks Language

transform MatrixMultiply
from A[c,h], B[w,c]
to AB[w,h]
{
    // Base case, compute a single element
    to(AB.cell(x,y) out)
    from(A.row(y) a, B.column(x) b) {
        out = dot(a, b);
    }
}

// Recursively decompose in c
to(AB ab)
from(A.region(0, 0, c/2, h ) a1,
    A.region(c/2, 0, c, h ) a2,
    B.region(0, 0, w, c/2) b1,
    B.region(0, c/2, w, c ) b2) {
    ab = MatrixAdd(MatrixMultiply(a1, b1),
                   MatrixMultiply(a2, b2));
}

• Implicitly parallel description

• Algorithmic choice
transform MatrixMultiply
from A[c,h], B[w,c]
to AB[w,h]
{
    // Base case, compute a single element
    to(AB.cell(x,y) out)
    from(A.row(y) a, B.column(x) b) {
        out = dot(a, b);
    }

    // Recursively decompose in w
    to(AB.region(0, 0, w/2, h ) ab1,
       AB.region(w/2, 0, w, h ) ab2)
    from( A a,
          B.region(0, 0, w/2, c ) b1,
          B.region(w/2, 0, w, c ) b2) {
        ab1 = MatrixMultiply(a, b1);
        ab2 = MatrixMultiply(a, b2);
    }

    // Recursively decompose in c
    to(AB ab)
    from(A.region(0, 0, c/2, h ) a1,
         A.region(c/2, 0, c, h ) a2,
         B.region(0, 0, w, c/2) b1,
         B.region(0, c/2, w, c ) b2) {
        ab = MatrixAdd(MatrixMultiply(a1, b1),
                        MatrixMultiply(a2, b2));
    }
}
PetaBricks Language

**transform** MatrixMultiply
**from** A[c,h], B[w,c]
**to** AB[w,h]

{  // Base case, compute a single element
     to(AB.cell(x,y) out)
     from(A.row(y) a, B.column(x) b) {
         out = dot(a, b);
     }

  // Recursively decompose in c
  to(AB ab)
  from(A.region(0, 0, c/2, h ) a1,
       A.region(c/2, 0, c, h ) a2,
       B.region(0, 0, w, c/2) b1,
       B.region(0, c/2, w, c ) b2) {
      ab = MatrixAdd(MatrixMultiply(a1, b1),
                      MatrixMultiply(a2, b2));
  }

  // Recursively decompose in w
  to(AB.region(0, 0, w/2, h ) ab1,
     AB.region(w/2, 0, w, h ) ab2)
  from( A a,
         B.region(0, 0, w/2, c ) b1,
         B.region(w/2, 0, w, c ) b2) {
     ab1 = MatrixMultiply(a, b1);
     ab2 = MatrixMultiply(a, b2);
  }

  // Recursively decompose in h
  to(AB.region(0, 0, w, h/2) ab1,
     AB.region(0, h/2, w, h ) ab2)
  from(A.region(0, 0, c, h/2) a1,
       A.region(0, h/2, c, h ) a2,
       B b) {
     ab1=MatrixMultiply(a1, b);
     ab2=MatrixMultiply(a2, b);
  }
}
PetaBricks Language

transform Strassen
  from A11[n,n], A12[n,n], A21[n,n], A22[n,n],
      B11[n,n], B12[n,n], B21[n,n], B22[n,n]
  through M1[n,n], M2[n,n], M3[n,n], M4[n,n], M5[n,n], M6[n,n], M7[n,n]
  to   C11[n,n], C12[n,n], C21[n,n], C22[n,n]
  {   to(M1 m1) from(A11 a11, A22 a22, B11 b11, B22 b22) using(t1[n,n],
              t2[n,n]) {
    MatrixAdd(t1, a11, a22);
    MatrixAdd(t2, b11, b22);
    MatrixMultiplySqr(m1, t1, t2);
  }
  to(M2 m2) from(A21 a21, A22 a22, B11 b11) using(t1[n,n]) {
    MatrixAdd(t1, a21, a22);
    MatrixMultiplySqr(m2, t1, b11);
  }
  to(M3 m3) from(A11 a11, B12 b12, B22 b22) using(t1[n,n]) {
    MatrixSub(t2, b12, b22);
    MatrixMultiplySqr(m3, a11, t2);
  }
  to(M4 m4) from(A22 a22, B21 b21, B11 b11) using(t1[n,n]) {
    MatrixSub(t2, b21, b11);
    MatrixMultiplySqr(m4, a22, t2);
  }
  to(M5 m5) from(A11 a11, A12 a12, B22 b22) using(t1[n,n]) {
    MatrixAdd(t1, a11, a12);
    MatrixMultiplySqr(m5, t1, b22);
  }
  to(M6 m6) from(A21 a21, A11 a11, B11 b11, B12 b12)
    using(t1[n,n], t2[n,n]) {
    MatrixSub(t1, a21, a11);
    MatrixAdd(t2, b11, b12);
    MatrixMultiplySqr(m6, t1, t2);
  }
  to(M7 m7) from(A12 a12, A22 a22, B21 b21, B22
    b22) using(t1[n,n], t2[n,n]) {
    MatrixSub(t1, a12, a22);
    MatrixAdd(t2, b21, b22);
    MatrixMultiplySqr(m7, t1, t2);
  }
  to(C11 c11) from(M1 m1, M4 m4, M5 m5, M7 m7) {
    MatrixAddAddSub(c11, m1, m4, m7, m5);
  }
  to(C12 c12) from(M3 m3, M5 m5) {
    MatrixAdd(c12, m3, m5);
  }
  to(C21 c21) from(M2 m2, M4 m4) {
    MatrixAdd(c21, m2, m4);
  }
  to(C22 c22) from(M1 m1, M2 m2, M3 m3, M6 m6) {
    MatrixAddAddSub(c22, m1, m3, m6, m2);
  }
}
Language Support for Algorithmic Choice

- Algorithmic choice is the key aspect of PetaBricks
- Programmer can define multiple rules to compute the same data
- Compiler re-use rules to create hybrid algorithms
- Can express choices at many different granularities
Synthesized Outer Control Flow

- Outer control flow synthesized by compiler
- Another choice that the programmer should not make
  - By rows?
  - By columns?
  - Diagonal? Reverse order? Blocked?
  - Parallel?
- Instead programmer provides explicit producer-consumer relations
- Allows compiler to explore choice space
Outline

• Target Audience
• Why We Need a New Language
• PetaBricks Language
• **PetaBricks Compiler**
• PetaBricks Performance
• Variable Accuracy
transform RollingSum
from A[n]
to B[n]
{
    // rule 0: use the previously computed value
    B.cell(i) from (A.cell(i) a, B.cell(i-1) leftSum) {
        return a + leftSum;
    }
    // rule 1: sum all elements to the left
    B.cell(i) from (A.region(0, i) in) {
        return sum(in);
    }
}
Compilation Process

- Applicable Regions
- Choice Grids
- Choice Dependency Graphs
Applicable Regions

// rule 0: use the previously computed value
B.cell(i) from (A.cell(i) a, B.cell(i-1) leftSum) {
    return a + leftSum;
}
Rule is Applicable in the Region: 1 ≤ i < n

// rule 1: sum all elements to the left
B.cell(i) from (A.region(0, i) in) {
    return sum(in);
}
Rule is Applicable in the Region: 0 ≤ i < n
Choice Grids

- Divide data space into symbolic regions with common sets of choices
- In this simple example:
  - A: Input (no choices)
  - B: [0; 1) = rule 1
  - B: [1; n) = rule 0 or rule 1
- Applicable regions map *rules* $\rightarrow$ *symbolic data*
- Choice grids map *symbolic data* $\rightarrow$ *rules*
Choice Dependency Graphs

- Adds dependency edges between symbolic regions
- Edges annotated with directions and rules
- Many compiler passes on this IR to:
  - Simplify complex dependency patterns
  - Add choices
PetaBricks Flow

1. PetaBricks source code is compiled
2. An autotuning binary is created
3. Autotuning occurs creating a choice configuration file
4. Choices are fed back into the compiler to create a static binary
Autotuning

• Based on two building blocks:
  – A genetic tuner
  – An n-ary search algorithm

• Flat parameter space

• Compiler generates a dependency graph describing this parameter space

• Entire program tuned from bottom up
Outline

- Target Audience
- Why We Need a New Language
- PetaBricks Language
- PetaBricks Compiler
- PetaBricks Performance
- Variable Accuracy
Sort

Time

0.000 0.002 0.004 0.006 0.008 0.010

Size

0 200 400 600 800 1000 1200 1400 1600 1800 2000

Insertion Sort
Quick Sort
Merge Sort
Radix Sort
Sort

- Insertion Sort
- Quick Sort
- Merge Sort
- Radix Sort
- Autotuned
## Future Proofing Sort

<table>
<thead>
<tr>
<th>System</th>
<th>Cores used</th>
<th>Scalability</th>
<th>Algorithm Choices (w/ switching points)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile Core 2 Duo Mobile</td>
<td>2 of 2</td>
<td>1.92</td>
<td>IS(150) 8MS(600) 4MS(1295) 2MS(38400) QS(1)</td>
</tr>
<tr>
<td>Xeon 1-way Xeon E7340 (2 x 4 core)</td>
<td>1 of 8</td>
<td>-</td>
<td>IS(75) 4MS(98) RS(1)</td>
</tr>
<tr>
<td>Xeon 8-way Xeon E7340 (2 x 4 core)</td>
<td>8 of 8</td>
<td>5.69</td>
<td>IS(600) QS(1420) 2MS(1)</td>
</tr>
<tr>
<td>Niagara Sun Fire T200</td>
<td>8 of 8</td>
<td>7.79</td>
<td>16MS(75) 8MS(1461) 4MS(2400) 2MS(1)</td>
</tr>
</tbody>
</table>

### Trained On

<table>
<thead>
<tr>
<th>Run On</th>
<th>Mobile</th>
<th>Xeon 1-way</th>
<th>Xeon 8-way</th>
<th>Niagara</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile</td>
<td>-</td>
<td>1.09x</td>
<td>1.67x</td>
<td>1.47x</td>
</tr>
<tr>
<td>Xeon 1-way</td>
<td>1.61x</td>
<td>-</td>
<td>2.08x</td>
<td>2.50x</td>
</tr>
<tr>
<td>Xeon 8-way</td>
<td>1.59x</td>
<td>2.14x</td>
<td>-</td>
<td>2.35x</td>
</tr>
<tr>
<td>Niagara</td>
<td>1.12x</td>
<td>1.51x</td>
<td>1.08x</td>
<td>-</td>
</tr>
</tbody>
</table>
Eigenvector Solve

![Eigenvector Solve Graph]

- Bisection
- DC
- QR

Size vs. Time Graph
Eigenvector Solve

![Graph showing performance of different methods for solving eigenvector problems as a function of size. The x-axis represents the size, and the y-axis represents the time. The methods compared are Bisection, DC, QR, and Autotuned.](image)
Outline

• Target Audience
• Why We Need a New Language
• PetaBricks Language
• PetaBricks Compiler
• PetaBricks Performance
• **Variable Accuracy**
Poisson's Equation: Variable Accuracy Benchmark

- Accuracy level expressed as a template parameter
- Autotuner exploits variable accuracy in a general way
- Choices:
  - Direct solve
  - Jacobi iteration
  - Successive over relaxation
  - Multigrid
Dynamic programming technique for autotuning Multigrid

- Candidate algorithms each have measured time/accuracy
Dynamic programming technique for autotuning Multigrid

- Candidate algorithms each have measured time/accuracy
- Optimal frontier of algorithms not dominated by others
Dynamic programming technique for autotuning Multigrid

- Candidate algorithms each have measured time/accuracy
- Optimal frontier of algorithms not dominated by others
- Partition accuracy space into discrete levels
Dynamic programming technique for autotuning Multigrid

- Candidate algorithms each have measured time/accuracy
- Optimal frontier of algorithms not dominated by others
- Partition accuracy space into discrete levels
Dynamic programming technique for autotuning Multigrid

- Candidate algorithms each have measured time/accuracy
- Optimal frontier of algorithms not dominated by others
- Partition accuracy space into discrete levels
Dynamic programming technique for autotuning Multigrid

- Candidate algorithms each have measured time/accuracy
- Optimal frontier of algorithms not dominated by others
- Partition accuracy space into discrete levels
- Base space of candidate algorithms on optimal algorithms from coarser resolution
Autotuned V-cycle shapes

- Arrow = direct solve; Dot = SOR iteration; Line = Coarsen/Refine grid
Poisson

Matrix Size

Time

Direct
Jacobi
SOR
Multigrid

Poisson

Matrix Size

Time

Direct
Jacobi
SOR
Multigrid
Poisson

Matrix Size

Time

Direct
Jacobi
SOR
Multigrid
Autotuned
Conclusion

• Autotuning is a powerful and promising technique (dah!)
• Autotuning can be used to eliminate the burden of obtaining performance from the average programmer
• However, without language support, it is hard to harness the full power of autotuning

• We are looking for collaborators
  – To target Petrabricks to the expert programmers

http://projects.csail.mit.edu/petabricks