Compiler Techniques for Optimizing Dense Matrix Multiplication on a Many-Core Architecture

Elkin Garcia\textsuperscript{1} Ioannis E. Venetis\textsuperscript{2} Rishi Khan\textsuperscript{3} Kelly Livingston\textsuperscript{1} Guang R. Gao\textsuperscript{1}

Computer Architecture and Parallel Systems Laboratory
Department of Electrical and Computer Engineering
University of Delaware, Newark 19716, U.S.A.
\{egarcia,ggao\}@capsl.udel.edu

Department of Computer Engineering and Informatics
University of Patras, Rion 26500, Greece
venetis@ceid.upatras.gr

ET International, Newark 19711, U.S.A.
rishi@etinternational.com

November 3rd, 2010
Outline

Introduction
- Motivation
- Objectives

Background
- IBM Cyclops-64
- Classical MM Algorithms

Proposed Matrix Multiplication Algorithm
- Work Distribution
- Compiler Optimizations

Experimental Evaluation
- Results

Conclusions
Introduction
Motivation
Objectives

Background
IBM Cyclops-64
Classical MM Algorithms

Proposed Matrix Multiplication Algorithm
Work Distribution
Compiler Optimizations

Experimental Evaluation
Results

Conclusions
Traditional Parallel Programming Methodologies

**Goal:** Improve performance.

**Assumptions:** Cache based parallel systems.

**Strategies:** Cache tiling techniques exploit temporal locality.

- Tile size selection and padding.

- Data location and replacement in the cache is controlled by HW making fine control of these parameters difficult.

- Power consumption and chip die area constraints make increasing on-chip cache an untenable solution to the memory wall problem.
Traditional Parallel Programming Methodologies

**Goal:** Improve performance.
**Assumptions:** Cache based parallel systems.
**Strategies:** Cache tiling techniques exploit temporal locality.
- Tile size selection and padding.

- Data location and replacement in the cache is controlled by HW making fine control of these parameters difficult.
- Power consumption and chip die area constraints make increasing on-chip cache an untenable solution to the memory wall problem.
New many-core-on-a-chip Systems

- Software managed memory hierarchy.
  - The programmer has the control of data movement.
  - Save die area of hardware cache controllers and over-sized caches.
  - More flexibility and opportunities to improve performance.
  - The programming at this moment is more complicated.

- Example: IBM Cyclops-64 (C64).
New many-core-on-a-chip Systems

- Software managed memory hierarchy.
  - The programmer has the control of data movement.
  - Save die area of hardware cache controllers and over-sized caches.
  - More flexibility and opportunities to improve performance.
  - The programming at this moment is more complicated.

- Example: IBM Cyclops-64 (C64).

New methodologies for classical algorithmic problems are needed
What has been done?

Many well-known algorithms has been ported and optimized for many-core architectures applying and adapting strategies of cache-based parallel systems.

- Matrix Multiplication, LU Decomposition, FFT, etc.

- The optimizations for improving performance on cache-based parallel system are not necessarily feasible or convenient on software managed memory hierarchy systems.
- Memory access patterns reached by appropriate tiling substantially increase the performance of applications.
# What has been done?

Many well-known algorithms has been ported and optimized for many-core architectures applying and adapting strategies of cache-based parallel systems.

- Matrix Multiplication, LU Decomposition, FFT, etc.

<table>
<thead>
<tr>
<th>The optimizations for improving performance on cache-based parallel system are not necessarily feasible or convenient on software managed memory hierarchy systems.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory access patterns reached by appropriate tiling substantially increase the performance of applications.</td>
</tr>
</tbody>
</table>
Objectives

Propose a general methodology that provides a mapping of applications to software managed memory hierarchies. 3 strategies for increasing performance:

1. Balanced distribution of work among threads.

We used MM on C64 as a case of study because:

- It is simple: A basic MM is described by 3 for loops.
- It is memory and computational intensive: The basic MM is $O(m^3)$. 
Introduction
Motivation
Objectives

Background
IBM Cyclops-64
Classical MM Algorithms

Proposed Matrix Multiplication Algorithm
Work Distribution
Compiler Optimizations

Experimental Evaluation
Results

Conclusions
The IBM Cyclops-64 Architecture

- The complete C64 system is built out of tens of thousands of C64 processing nodes arranged in a 3-D mesh topology.
- Each processing node consists of a C64 chip, external DRAM, and a small amount of external interface logic.
- Execution on a C64 chip is non-preemptive and there is no hardware virtual memory manager.
Classical Matrix Multiplication Algorithms

- Decrease the naïve complexity of $O(m^3)$: No architecture dependent (at all)
  - Strassen’s algorithm: $O(m^{\log_2 7})$.
  - Coppersmith–Winograd algorithm: $O(m^{2.376})$.
- Efficient implementations: Architecture dependent.
  - Blocking Algorithms.
  - Explore the Architecture design space.
  - Example: Cannon’s Algorithm for 3D mesh.

Many-core architecture design space has not yet been explored in detail.
Classical Matrix Multiplication Algorithms

- Decrease the naïve complexity of $O(m^3)$: No architecture dependent (at all)
  - Strassen’s algorithm: $O(m^{\log_7})$.
  - Coppersmith–Winograd algorithm: $O(m^{2.376})$.
- Efficient implementations: Architecture dependent.
  - Blocking Algorithms.
  - Explore the Architecture design space.
  - Example: Cannon’s Algorithm for 3D mesh.

Many-core architecture design space has not yet been explored in detail.
Introduction
Motivation
Objectives

Background
IBM Cyclops-64
Classical MM Algorithms

Proposed Matrix Multiplication Algorithm
Work Distribution
Compiler Optimizations

Experimental Evaluation
Results

Conclusions
Dense Matrix Multiplication

- $A \times B = C$ each of size $m \times m$ using algorithms of running time $O(m^3)$ using $P$ Processors.
- Related sources that cause poor performance in many-core architectures:
  1. Inefficient or unnecessary synchronizations.
  2. Unbalanced work between threads.
  3. Latency due to accessing slower memory levels or other kind of instructions.
  4. Stalls due to arbitration of shared resources.
- There is a trade-off between synchronization and work-balanced.
Work Distribution

• For MM, each element $c_{i,j} \in C$ can be calculated independently.
  • Synchronizations are not needed.

• Optimal partition.
  Blocks size: $\frac{m^2}{P}$.

• Constrains:
  • Number of elements in each block has to be integer.
  • Blocks are rectangular.
Optimal Work Distribution

Matrix $C$ divided in $P$ blocks $C'$. $q_1 \cdot q_2 = P$

- Minimize the difference between:
  - Maximum tile size $\left\lceil \frac{m}{q_1} \right\rceil \cdot \left\lceil \frac{m}{q_2} \right\rceil$ AND Optimal tile size $\frac{m^2}{P}$.
  - Optimum is reached when $q_1 = q_2 = \sqrt{P}$.
- In practice, we can turn off some processors if the maximum tile size can be decreased.
  - Example: $P$ is prime.
High Cost Memory Operations

- High bandwidth of on-chip memory in many-core architectures is not enough.
- Programmer can take advantage of the new opportunities provided by software-managed memory hierarchies.
- Goal: Minimize the number of memory operations ($LD$ and $ST$) between a bigger but slower memory level (SRAM) and a faster but smaller one (Registers).
- That may are function of:
  - The problem ($\Lambda$).
  - The number of processors ($P$).
  - The tile parameters ($L$).
  - The sequence of traversing tiles ($S$).
  - The size of the faster memory $R_{max}$
Optimization Problem Formulation

\[
\min_{L,S} \quad LD(\Lambda, P, L, S) + ST(\Lambda, P, L, S)
\]

s.t. \quad R(\Lambda, P, L, S) \leq R_{\text{max}}

- \Lambda = MM.
- \(1 \leq P \leq P_{\text{max}}\).
- \(L = \{L_1, L_2\}\).
- \(S = \{S_1, S_2, S_3, S_4, S_5, S_6\}\)
Optimization Problem Formulation

\[
\min_{L, S} \quad LD (\Lambda, P, L, S) + ST (\Lambda, P, L, S) \\
\text{s.t.} \quad R (\Lambda, P, L, S) \leq R_{\text{max}}
\]

- \( \Lambda = MM \).
- \( 1 \leq P \leq P_{\text{max}} \).
- \( L = \{L_1, L_2\} \).
- \( S = \{S_1, S_2, S_3, S_4, S_5, S_6\} \)
Matrices $A$, $B$ and $C$ are partitioned in blocks $A'$, $B'$ and $C'$ of sizes $n \times m$, $m \times n$ and $n \times n$.  

$A'$, $B'$ and $C'$ are divided in tiles $A'_{i,j}$, $B'_{i,j}$ and $C'_{i,j}$ of sizes $L_2 \times L_1$, $L_1 \times L_2$ and $L_2 \times L_2$.  

**Proposed Matrix Multiplication Algorithm**

```plaintext
for i=1 to n/L2
    for j=1 to n/L2
        for k=1 to m/L1
            C'_{i,j} += A'_{i,k} * B'_{k,j}
```

```plaintext
MM for a block C'
```
6 possible schemes of traversing tiles that produce 2 sequences.

- Case 1: Reuse tile $C'_{i,j}$.
- Case 2: Reuse tile $A'_{i,j}$ (or $B'_{i,j}$).
Optimization Problem for MM

\[
\begin{align*}
\min_{L \in \{L_1, L_2\}, S \in \{S_1, S_2\}} & \quad f(m, P, L, S) = \begin{cases} 
\frac{2}{L_2} m^3 + m^2 & \text{if } S = S_1 \\
\left(\frac{2}{L_1} + \frac{1}{L_2}\right) m^3 + \left(\sqrt{P} - 1\right) m^2 & \text{if } S = S_2
\end{cases} \\
\text{s.t.} & \quad 2L_1 L_2 + L_2^2 \leq R_{\text{max}}
\end{align*}
\]

Analytical solution if \( P \geq 4 \) using KKT multipliers. Solution was found by branch and bound (1 iter.):

\[ L_1 = 1, \quad L_2 = \left[\sqrt{1 + R_{\text{max}}} - 1\right] \]
Optimization Problem for MM

\[
\min_{L \in \{L_1, L_2\}, S \in \{S_1, S_2\}} f(m, P, L, S) = \begin{cases} \frac{2}{L_2} m^3 + m^2 & \text{if } S = S_1 \\ \left(\frac{2}{L_1} + \frac{1}{L_2}\right) m^3 + \left(\sqrt{P} - 1\right) m^2 & \text{if } S = S_2 \end{cases}
\]

s.t. \quad 2L_1L_2 + L_2^2 \leq R_{\text{max}}

Analytical solution if \( P \geq 4 \) using KKT multipliers. Solution was found by branch and bound (1 iter.):
\[
L_1 = 1, \quad L_2 = \left\lfloor \sqrt{1 + R_{\text{max}}} - 1 \right\rfloor
\]

\[\text{m=12; P=4; Rmax=15} \quad \text{n=6} \quad \text{L1=1; L2=3}\]
Real example: Cyclops-64

- Number of registers: 63
- Other Register used: 6
- $R_{max} = 57$
- $L_1 = 1$ and $L_2 = 6$
- Other tiling strategies that fully utilizes the registers:
  - Inner Product: $L_1 = 28$ and $L_2 = 1$
  - Square Tiling: $L_1 = 4$ and $L_2 = 4$

Table: Number of memory operation for different tiling strategies

<table>
<thead>
<tr>
<th>Memory Operations</th>
<th>Inner Product</th>
<th>Square</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>$2m^3/m^2$</td>
<td>$\frac{1}{2}m^3/m^2$</td>
<td>$\frac{1}{3}m^3/m^2$</td>
</tr>
<tr>
<td>Stores</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Real example: Cyclops-64

- Number of registers: 63
- Other Register used: 6
- $R_{max} = 57$
- $L_1 = 1$ and $L_2 = 6$
- Other tiling strategies that fully utilizes the registers:
  - Inner Product: $L_1 = 28$ and $L_2 = 1$
  - Square Tiling: $L_1 = 4$ and $L_2 = 4$

Table: Number of memory operation for different tiling strategies

<table>
<thead>
<tr>
<th>Memory Operations</th>
<th>Inner Product</th>
<th>Square</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$2m^3$</td>
<td>$\frac{1}{2}m^3$</td>
<td>$\frac{1}{3}m^3$</td>
</tr>
<tr>
<td>Loads</td>
<td>$m^2$</td>
<td>$m^2$</td>
<td>$m^2$</td>
</tr>
<tr>
<td>Stores</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Real example: Cyclops-64

- Number of registers: 63
- Other Register used: 6
- $R_{max} = 57$
- $L_1 = 1$ and $L_2 = 6$
- Other tiling strategies that fully utilizes the registers:
  - Inner Product: $L_1 = 28$ and $L_2 = 1$
  - Square Tiling: $L_1 = 4$ and $L_2 = 4$

**Table:** Number of memory operation for different tiling strategies

<table>
<thead>
<tr>
<th>Memory Operations</th>
<th>Inner Product</th>
<th>Square</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>$2m^3$</td>
<td>$\frac{1}{2}m^3$</td>
<td>$\frac{1}{3}m^3$</td>
</tr>
<tr>
<td>Stores</td>
<td>$m^2$</td>
<td>$m^2$</td>
<td>$m^2$</td>
</tr>
</tbody>
</table>
Keep in mind for a Register Tiling Design

1. Goal: Maximize Reuse of data in registers (Diminish Memory Operations)
2. Register Allocation.
   - Spilling.
   - Scratchpad Memory.
Instruction Selection and Instruction Scheduling

Multiple Load (ldm) and Multiple Store (stm)
- Normal load instruction issues one data transfer request per element while the special one issues one request each 64-byte boundary.
- Useful for load tiles of A (6x1) and B (1x6) with A in column-major order and B in row-major order.

Instruction Scheduling
- Interleaving of independent instruction to alleviate stalls.
  - Memory Operations.
  - Data Operations.
    - Floating point Operations.
    - Integer Operations.
Diminish/Hide Latencies of Instructions

- Data dependencies imposes partial ordering on execution.
- Instruction Scheduling hides or diminishes the cost of stalls produces by large latencies. (e.g. ldm, divs, rems, mull).
  - It could require Register Reallocation.
- Data Prefetching and Loop Unrolling.
  - Partial hiding of latencies will still hurt the performance. We cannot reach peak performance if we don’t hide ALL latencies.
  - It definitely requires Retiling, Reallocation, Rescheduling.
Data Prefetching and Loop Unrolling

- Guarantee total hiding of latencies.

\[ S1: \ c[1..L_1][1..L_2] = 0 \]
\[ S2: \ for \ k = 1 \ \text{to} \ m, \ k \ + \ + \]
\[ S3: \ a[1..L_1][1] = A[i..i + L_1][k] \]
\[ S4: \ b[1][1..L_2] = B[k][j..j + L_2] \]
\[ S5: \ c[1..L_1][1..L_2] + = a[1..L_1][1] \times b[1][1..L_2] \]
\[ S : \ \text{end for} \]
\[ S6: \ C[i..i + L_1][j..j + L_2] = c[1..L_1][1..L_2] \]

\( C \) tile calculation of size \( L_1 \times L_2 \) without loop unrolling

\[ S1 : \ c[1..L_1][1..L_2] = 0 \]
\[ S2 : \ a[1..L_1][1] = A[i..i + L_1][k] \]
\[ S3 : \ b[1][1..L_2] = B[k][j..j + L_2] \]
\[ S4 : \ for \ k = 1 \ \text{to} \ m, \ k \ + \ + \]
\[ S5 : \ a[1..L_1][2] = A[i..i + L_1][k + 1] \]
\[ S6 : \ b[2][1..L_2] = B[k + 1][j..j + L_2] \]
\[ S7 : \ c[1..L_1][1..L_2] + = a[1..L_1][1] \times b[1][1..L_2] \]
\[ S7 : \ k \ + \ +, \ \text{if} \ k = = m \ \text{then break} \]
\[ S9 : \ a[1..L_1][1] = A[i..i + L_1][k + 1] \]
\[ S10 : \ b[1][1..L_2] = B[k + 1][j..j + L_2] \]
\[ S11 : \ c[1..L_1][1..L_2] + = a[1..L_1][2] \times b[2][1..L_2] \]
\[ S : \ \text{end for} \]
\[ S12 : \ C[i..i + L_1][j..j + L_2] = c[1..L_1][1..L_2] \]

\( C \) tile calculation of size \( L_1 \times L_2 \) with loop unrolling

The price is to increase the register pressure.
Introduction
   Motivation
   Objectives

Background
   IBM Cyclops-64
   Classical MM Algorithms

Proposed Matrix Multiplication Algorithm
   Work Distribution
   Compiler Optimizations

Experimental Evaluation
   Results

Conclusions
**Partitioning**

Matrix Size $100 \times 100$

- **Partition1**: Tile size is around the optimum $\left\lfloor \frac{m}{q_1} \right\rfloor \cdot \left\lfloor \frac{m}{q_2} \right\rfloor$ but it does **NOT** minimize the maximum tile size.

- **Partition2**: Minimize the maximum tile size but it does **NOT** distribute sizes uniformly.

- **Partition3**: Optimum partitioning and distribution.

Matrix Size $488 \times 488$
Impact of each optimization on the performance

$m_{SRAM} = 488$, $m_{DRAM} = 5280$

1. Base Parallel Version.
2. +Optimized partitioning.
4. +Multiple load/store inst. (Man.).
5. +Instruction Sched.(Man.).
6. +Dynamic Scheduling (Man.).
7. +Data Prefetching (Man.).
8. +Instruction Prefetching (Man.).
9. +Operands on DRAM.
10. +Dynamic Percolation (Man.)
11. +Optimized MemCpy and MemCpyTranspose (Man.)

Power consumption: 66W → 993 MFLOPS/W

Green500list: Most energy-efficient supercomputers in the world

- Top 1-3: 722.98 MFLOPS/W (Nov’09).
- Top 1-3: 773.38 MFLOPS/W
- Top 4-5: 458.33 MFLOPS/W
- Top 4: 492.64 MFLOPS/W (Jun’10).
Impact of each optimization on the performance

$m_{SRAM} = 488, m_{DRAM} = 5280$

1. Base Parallel Version.
2. +Optimized partitioning.
4. +Multiple load/store inst. (Man.).
5. +Instruction Sched.(Man.).
6. +Dynamic Scheduling (Man.).
7. +Data Prefetching (Man.).
8. +Instruction Prefetching (Man.).
9. +Operands on DRAM.
10. +Dynamic Percolation (Man.)
11. +Optimized MemCpy and MemCpyTranspose (Man.)

Power consumption:
66W $\rightarrow$ 993 MFLOPS/W

Green500list: Most energy-efficient supercomputers in the world

- Top 1-3: 722.98 MFLOPS/W
- Top 4-5: 458.33 MFLOPS/W  (Nov’09).
- Top 1-3: 773.38 MFLOPS/W
- Top 4: 492.64 MFLOPS/W  (Jun’10).
Impact of each optimization on the performance

\[ m_{SRAM} = 488, \quad m_{DRAM} = 5280 \]

1. Base Parallel Version.
2. +Optimized partitioning.
4. +Multiple load/store inst. (Man.).
5. +Instruction Sched.(Man.).
6. +Dynamic Scheduling (Man.).
7. +Data Prefetching (Man.).
8. +Instruction Prefetching (Man.).
9. +Operands on DRAM.
10. +Dynamic Percolation (Man.)
11. +Optimized MemCpy and MemCpyTranspose (Man.)

**Power consumption:**

66W → 993 MFLOPS/W

Green500list: Most energy-efficient supercomputers in the world

- Top 1-3: 722.98 MFLOPS/W (Nov’09).
- Top 1-3: 773.38 MFLOPS/W
- Top 4-5: 458.33 MFLOPS/W
- Top 4: 492.64 MFLOPS/W (Jun’10).
Introduction
  Motivation
  Objectives

Background
  IBM Cyclops-64
  Classical MM Algorithms

Proposed Matrix Multiplication Algorithm
  Work Distribution
  Compiler Optimizations

Experimental Evaluation
  Results

Conclusions
Conclusions

- Software-managed memory hierarchies provide more flexibility and opportunities for increasing performance that have not been explored at all.
- Compiler Optimizations at register level are essential for increasing performance. Most of them are highly correlated.
- Compiler optimizations applied provide evidence of the power efficiency of C64: power consumption measurements show a maximum efficiency of 993 MFLOPS/W for the problem under consideration.
- Dynamic strategies deserve more attention. This case of study has inspired promising techniques such as the codelet model.
Future work

- Apply this methodology to other linear algebra algorithmic problems like matrix inversion and linear solver (Linpack). Expand to multiple chips.
- How can we apply these optimizations for increasing energy efficiency? Does maximum performance imply maximum energy efficiency?
Thank you