High-Performance Vectorization on GPU Cluster by MVAPICH2

Di Zhao
zhao.1029@osu.edu
Ohio State University

MVAPICH User Group (MUG) Meeting, August 26-27 2013, Columbus Ohio
# Introduction to Accelerator

<table>
<thead>
<tr>
<th></th>
<th>Nvidia Kepler K20X</th>
<th>Intel Xeon Phi 7120</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Launch Date</strong></td>
<td>November 2012</td>
<td>Q2 2013</td>
</tr>
<tr>
<td><strong>Processor</strong></td>
<td>14 Streaming Multiprocessors</td>
<td>61 Pentium x86 cores</td>
</tr>
<tr>
<td><strong>Per-processor Concurrency</strong></td>
<td>192 CUDA cores (SIMT)</td>
<td>4 hyperthreads ×8(512 bit)× SIMD units</td>
</tr>
<tr>
<td><strong>Total Nominal Concurrency</strong></td>
<td>2688 (14×192)</td>
<td>1952 (61×4×8)</td>
</tr>
<tr>
<td><strong>Acceleration Techniques</strong></td>
<td>Vectorization, Shared memory, OpenACC</td>
<td>Vectorization</td>
</tr>
</tbody>
</table>
Introduction

Vectorization: a data-level parallelism, vectorization is the process of converting an algorithm from a scalar implementation, which does an operation one pair of operands a time, to a vector process, where a single instruction can refer to a vector (series of adjacent values.)
Introduction
Vectorization

- **loop vectorization**: independent loop, loop with if conditions, *etc.*;
- matrix tile: element tile, row/column tile, matrix tile, *etc.*;
- index operations: idx2val, val2idx, find, fetch/write by index, *etc.*;
- element-wise binary operation;
- logical operation: all, any, logical, *etc.*;
- *etc.*;
Application: all fields

- high-performance optimizer;
- high-performance integration;
- high-performance sampler;
- high-performance tri-diagonal solver, halo exchange;
- high-dimensional look-up table;
- etc.;
- Building blocks for high-level fields such as machine learning and artificial intelligence, computer graphics, computer vision, simulation, etc.;
- Usually defined on single GPU, research needed for multi-GPU and GPU cluster;
### Introduction

**Vectorization: for-loop**

```plaintext
for i = 1:I
    operation (i);
end
```

<table>
<thead>
<tr>
<th></th>
<th>Computational Cost</th>
<th>Memory Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>for-loop</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>vectorization</td>
<td>1</td>
<td>I</td>
</tr>
</tbody>
</table>
for $i_1 = 1:I_1$
  for $i_2 = 1:I_2$
    operation $(i_1,i_2)$;
  end
end

operation$(I_1*I_2)$

<table>
<thead>
<tr>
<th></th>
<th>Computational Cost</th>
<th>Memory Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>for-loop</td>
<td>$I_1*I_2$</td>
<td>1</td>
</tr>
<tr>
<td>vectorization</td>
<td>1</td>
<td>$I_1*I_2$</td>
</tr>
</tbody>
</table>

vectorization of 2-nested loop applications: find (for data compression), bin locating, etc.;
Introduction

Vectorization: $k$-nested for-loop

for $i_1 = 1: I_1$
  for $i_2 = 1: I_2$
    ●●●
    for $i_k = 1: I_k$
      operation $(i_1, i_2, \ldots, i_k)$;
    end
  end
end

operation$(I_1 \ast I_2 \cdots I_k)$

<table>
<thead>
<tr>
<th></th>
<th>Computational Cost</th>
<th>Memory Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>for-loop</td>
<td>$I_1 \ast I_2 \cdots I_k$</td>
<td>1</td>
</tr>
<tr>
<td>vectorization</td>
<td>1</td>
<td>$I_1 \ast I_2 \cdots I_k$</td>
</tr>
</tbody>
</table>

vectorization of $k$-nested loop applications: building high-dimensional look-up table
Introduction

Vectorization: *k*-nested for-loop

- For nested for-loop, memory cost increases Exponentially!
- For applications such as high-dimensional approximation, large $I_1 \times I_2 \cdots \times I_k$ is needed for high accuracy;
- For applications such as two-dimensional approximation, even two-dimensional $I_1 \times I_2$ is too big to fit in GPU’s global memory;
- Best performance of kernel execution comes from suitable setting the number of blocks and the number of threads per block, operation on full memory usually leads to bad performance;
Introduction

GPU-cluster Vectorization: 2-nested loop

for \( i_1 = 1 : I_1 \)
for \( i_2 = 1 : I_2 \)
    operation \((i_1, i_2)\);
end
end

for \( i_2 = 1 : I_2 \)
    operation \((i_1, i_2)\);
end

operation \((I_1, I_2)\);

The Problem: vector size exceeds GPU’s memory capacity

The Solution: GPU-cluster Vectorization

\[ \text{op}(I_1, I_2) \]
\[ \text{op}(I_1, I_2) \]
\[ \text{op}(I_1, I_2) \]
Step 1: In master GPU, separation of the array \((I_1, I_2)\);
Step 2: Send the separated array \((I_1, I_2')\) by MPI_SCATTER;
Step 3: Vectorized operation \(\text{op} (I_1, I_2')\) in each GPU;
Step 4: Collect the partial results to master GPU by MPI_GATHER;
Step 5: In master GPU, assembly the partial results back to the array \((I_1, I_2)\);

In GPU-cluster vectorization, Step 1, Step 3 and Step 5 is done in single GPU;
In GPU-cluster vectorization, the size of \((I_1, I_2')\) usually hundreds of kb, since the number of GPU is limited;
The performance of GPU-cluster vectorization is decided by the two collective communications: MPI_SCATTER and MPI_GATHER, especially while GPU-cluster vectorization locates in iteration or time step;
MVAPICH2;
Vectorization of \(k\)-nested loop is more complicated in communication;
Methods

High-performance Optimizer: an example of vectorization on GPU Cluster by MVAPICH2

- Bin Locating
- Iterative Discrete Approximation
- High-performance Optimizer

MVAPICH2
Bin Locating: Two arrays $S$ and $D$: the array $S$ is a sequence, the array $D$ is data. For a element $d_i$ in $D$, find the range $s_i \leq d_j < s_{i+1}$ in array $S$;

Implementation: comparing element $d_i$ in $D$ against the sequence $S$ one by one;
Methods

Bin Locating

- 2-nested for-loop: inner loop and outer loop.
  Computational Cost: $I \times J$; memory cost: 1;
  
  ```
  for i = 1:I
    for j = 1:J
      if ( s(j) <= d(i) & d(j) < s(i+1) )
        bin(i) = j;
        break;
      end
    end
  end
  ```

- Vectorization of inner-loop. Computational Cost: $I$; memory cost: 1;
  
  ```
  for i = 1:I
    bin = sum( d(i) - S);
  end
  ```

This element is tiled to array with size $J$
Methods

Bin Locating

- Vectorization of outer-loop. Computational Cost: 1; memory cost: $I \times J$;

  \[
  \text{bin} = \text{sumrow} \left( \text{tile}(D, J') - \text{tile}(S, I) \right);
  \]

- Splitting the array tile($D$, $J'$) and tile($S$, $I$) across multi-GPUs by MPI_Scatter;

- The computation of comparison is done in each GPU;

- Reduction $\text{sumrow}$ is implemented by CUDA thrust;

- Partial results are gathered by MPI_Gather;

- The Master-Slave Paradigm: master node and master GPU;
Iterative Discrete Approximation approximates complicated continuous distributions by discrete random numbers;

Large portion of computation of Iterative Discrete Approximation is spent on bin locating;
Methods
Iterative Discrete Approximation

\[ f(x) \]
Methods
Iterative Discrete Approximation

- Generate random numbers of uniform distribution;
- Estimation the shape of the function $f(x)$;
- Take advantage of the estimated shape, discretely approximate the function $f(x)$ by iterations;
- Implemented by GPU cluster;
Methods

High-Performance Optimizer

For large-scale and complicated search space, single Iterative Discrete Approximation is efficient, parallelization techniques are applied to make multiple Iterative Discrete Approximation work together.

Domain Decomposition of Search Space: understanding vectorized bin locating in the language of optimization
The search space is equally separated into multiple sub-space, and sent to each GPU by MPI_SCATTER;

The search space keeps changing in each iteration;

Each GPU is responsible for a sub-space;

Each GPU generates local optimum;

Local optima is collected by MPI_ALLREDUCE to calculate global optimum for next iteration;
Methods
High-Performance Optimizer

- Random Number Generation
  - Estimation of $f(x)$, Sort, Normalization, Bin Construction
    - Sub-bin $i$ Random Number
      - Local Optimum, Iterative Discrete Approximation
    - Collective Communication
  - Sub-bin $j$ Random Number
    - Local Optimum, Iterative Discrete Approximation
Computational Results

The problem: Finding the maximum peak from all peaks

\[ \max_{x} f(x) = a \times x \times \text{abs}(\sin(bx)^c) \]

\[ f(x, y) = \text{abs}\left(\frac{x}{a} \times \frac{y}{b} \times \sin(x + y)\right) \]
Computational Results

Time Cost of the Fourth Iteration by MVAPICH2 v.s. MPICH2

array size: the size of multiple arrays, not completely equal to message size

OSC Oakley cluster: 16 NVIDIA Tesla M2070, Mellanox IB QDR MT26428 Adapter
Computational Results

Time Cost along with Iterations by MVAPICH2 v.s. MPICH2

OSC Oakley cluster: 16 NVIDIA Tesla M2070, Mellanox IB QDR MT26428 Adapter
## Computational Results

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Numbers of Peaks</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single GPU</td>
<td>1D 232</td>
<td>$10^{-8}$</td>
</tr>
<tr>
<td>Multi-GPU</td>
<td>1D 241,717</td>
<td>$10^{-8}$</td>
</tr>
<tr>
<td>GPU-Cluster</td>
<td>2D 317,038</td>
<td>$10^{-6}$</td>
</tr>
</tbody>
</table>

**The Performance of Optimizer on GPU Clusters by MVAPICH2**
Computational Results

Accuracy along with Number of GPUs (i) left up array size = 10000 (ii) right up array size = 1000 (iii) left down array size = 100 (iv) right down array size = 10
Conclusions

- MVAPICH2 significantly improves the performance of vectorization, leading to high-efficiency applications such as optimizer;
- MVAPICH2 based GPU cluster is the platform to improve parallel programming techniques such as vectorization;
- More research of MVAPICH2 should be applied to more vectorization techniques;
Thanks!