Fast Interpolation of Grid Data at a Non-Grid Point

Hiroshi Inoue
IBM Research – Tokyo
Interpolation from Grid Data

Goal: to make compute-intensive interpolation operation faster

- Input: values at grid points
- Output: estimated (interpolated) value at a non-grid point

Target workloads include:
- medical imaging
  - CT reconstruction
  - registration etc
- stencil applications
  - particle simulation etc
Contributions

- Developed an fast method to interpolate values from grid data at a non-grid point

- Evaluated with 3D Computed Tomography (CT) reconstruction benchmark (RabbitCT)
  - this technique itself can be applicable for other imaging and non-imaging applications
  - although we explain the technique using bi-linear interpolation in this talk, it is applicable for more accurate interpolation algorithms (See paper for detail)
CT Reconstruction Overview

- Input: a set of 2D projection images obtained from different angles (and geometry information for each image)
- Output: density values for voxels in a 3D volume

Example of a (C-arm) CT system

Example of output from a CT system

Source: https://en.wikipedia.org/wiki/CT_scan

Projection in CT Reconstruction

Flat-panel detector (capturing 2D projection images)

Projected point \((u, v)\)

3D volume containing the object to be imaged

Voxel at \((x, y, z)\)

Point source of X-ray

Flat-panel detector (capturing 2D projection images)
Baseline Reconstruction Algorithm Overview [2]

for each projection image $I_n$
  
  // reconstruction (projection)
  for $z = 0$ to $L-1$
      for $y = 0$ to $L-1$
          for $x = 0$ to $L-1$
              for each voxel in 3D volume
                  1) project voxel $(x,y,z)$ onto $I_n$
                  2) read values from surrounding four grid points
                  3) estimate value at projected point by interpolation
                  4) update density value of voxel $(x,y,z)$
          end
      end
  end
end
Baseline Reconstruction Algorithm Overview [2]

for each projection image \( I_n \)

\[
\begin{aligned}
// & \text{ reconstruction (projection)} \\
& \text{for } z = 0 \text{ to } L-1 \\
& \quad \text{for } y = 0 \text{ to } L-1 \\
& \quad \quad \text{for } x = 0 \text{ to } L-1 \\
& \quad \quad \quad \text{1) project voxel } (x,y,z) \text{ onto } I_n \\
& \quad \quad \quad \text{2) read values from surrounding four grid points} \\
& \quad \quad \quad \text{3) estimate value at projected point by interpolation} \\
& \quad \quad \quad \text{4) update density value of voxel } (x,y,z) \\
& \quad \quad \end{aligned}
\]

end

end

end
Interpolation from Grid Data at Non-Grid Point

pixel at \((i, j)\)

pixel at \((i+1, j)\)

projected point \((u, v)\)

pixel at \((i, j+1)\)

pixel at \((i+1, j+1)\)
Interpolation from Grid Data at Non-Grid Point

\[ p_n(i, j) \]

\[ p_n(i + 1, j) \]

\[ \hat{p}_n(u, v) \]

\[ p_n(i, j + 1) \]

\[ p_n(i + 1, j + 1) \]
Interpolation from Grid Data at Non-Grid Point

Bilinear interpolation

\[ \hat{p}_n(u, v) = (1 - \alpha)(1 - \beta)p_n(i, j) + \alpha(1 - \beta)p_n(i + 1, j) + (1 - \alpha)\beta p_n(i, j + 1) + \alpha \beta p_n(i + 1, j + 1) \]
Do we have any redundancy in this formula?

$$\hat{p}_n(u,v) = (1 - \alpha)(1 - \beta)p_n(i, j) + \alpha(1 - \beta)p_n(i + 1, j) + (1 - \alpha)\beta p_n(i, j + 1) + \alpha \beta p_n(i + 1, j + 1)$$

😊 Yes, we can simplify it if many interpolation operations are done in one grid

- In CT reconstruction, hundreds to thousands of interpolations are executed in one grid on average

➜ Now, think $p_n(i, j), p_n(i + 1, j), p_n(i, j + 1), p_n(i + 1, j + 1)$ as constant values
Do we have any redundancy in this formula?

\[ \hat{p}_n(u,v) = (1-\alpha)(1-\beta)p_n(i,j) + \alpha(1-\beta)p_n(i+1,j) + (1-\alpha)\beta p_n(i,j+1) + \alpha\beta p_n(i+1,j+1) \]

\[ = (1-\alpha-\beta+\alpha\beta) \cdot p_n(i,j) \]
\[ + (\alpha-\alpha\beta) \cdot p_n(i+1,j) \]
\[ + (\beta-\alpha\beta) \cdot p_n(i,j+1) \]
\[ + \alpha\beta \cdot p_n(i+1,j+1) \]

Group terms by the number of \( \alpha \) and \( \beta \)

\[ = \alpha\beta \cdot C_0(i,j) + \alpha \cdot C_1(i,j) + \beta \cdot C_2(i,j) + C_3(i,j) \]

\[ C_0(i,j) = p_n(i,j) - p_n(i+1,j) - p_n(i,j+1) + p_n(i+1,j+1) \]
Our Efficient Interpolation Method

\[
\hat{p}_n(u,v) = (1 - \alpha)(1 - \beta)p_n(i, j) + \alpha(1 - \beta)p_n(i + 1, j) + (1 - \alpha)(1 - \beta)p_n(i, j + 1) + \alpha(1 - \beta)p_n(i + 1, j + 1)
\]

\[
= \alpha(\beta \cdot C_0(i, j) + C_1(i, j)) + (\beta \cdot C_2(i, j) + C_3(i, j))
\]

- If we have these four coefficients \(C_0 - C_3\), we can compute this formula with only three multiply-and-add instructions!

\(C_0 - C_3\) are independent from \(\alpha\) and \(\beta\) (and hence \(x, y, z\))

- We can pre-compute them at run time before iterating voxels and store in memory

original (without pre-computation)

for each projection image

projection

end

with pre-computation

for each projection image

pre-computation

projection

end

In the paper, we group terms based on \(u\) and \(v\) instead of \(\alpha\) and \(\beta\) for further performance boost:

\[
\alpha = u - i = u - \lfloor u \rfloor \quad \beta = v - j = v - \lfloor v \rfloor
\]
In-memory Pre-computed Table

Pre-computed Table

\[
\begin{align*}
C_3 & \quad C_0 & \quad C_1 & \quad C_2 & \quad C_3 & \quad C_0 & \quad C_1 \\
\end{align*}
\]

for \((i, j-1)\) \quad for \((i, j)\) \quad for \((i, j+1)\)

- 😞 total size of pre-computed table is 4x larger than original data
- 😊 need to read from only one cache line (w/ one aligned vector load)

Original data (Projection image) ➔ 😞 need to read from two cache lines

\[
\begin{align*}
\ldots \; p_n(i, j-1) \; p_n(i, j) \; p_n(i, j+1) \; p_n(i, j+2) \; p_n(i, j+3) \; p_n(i, j+4) \; p_n(i, j+5) \; \ldots \\
\ldots \; p_n(i+1, j-1) \; p_n(i+1, j) \; p_n(i+1, j+1) \; p_n(i+1, j+2) \; p_n(i+1, j+3) \; p_n(i+1, j+4) \; p_n(i+1, j+5) \; \ldots
\end{align*}
\]
Overall Algorithm with Pre-Computation

for each projection image $I_n$

// pre-computation
for $i = 0$ to $S_x-1$
    for $j = 0$ to $S_y-1$
        calculate and store coefficients $C_0$ to $C_3$ for pixel $(i, j)$
    end
end

// reconstruction (projection)
for $x = 0$ to $L-1$
    for $y = 0$ to $L-1$
        for $z = 0$ to $L-1$
            1) project voxel $(x, y, z)$ onto $I_n$
            2) read coefficients $C_0$ to $C_3$ from pre-computed table
            3) estimate value at projected point by interpolation
            4) update density value of voxel $(x, y, z)$
        end
    end
end
Performance Evaluation with RabbitCT

- RabbitCT is a framework for evaluating 3D CT reconstruction on performance and accuracy.
- It includes:
  - benchmark driver
  - reference implementations of the backprojection algorithm
  - input data (a C-arm CT dataset of a rabbit)
    - 496 projection images of 1248x960 pixels associated with transformation matrixes
- Output data is 3-D images of $256^3$ mm$^3$ space, $L^3 = 128^3$, $256^3$, $512^3$, $1024^3$ voxels, 12-bit value per voxel
System used for evaluations

- 2-socket POWER8 3.69 GHz
  - 20 cores in total (5 cores / NUMA node)
  - 8 SMT threads / core
- 256 GB system memory
- Ubuntu Linux 14.10 for Little Endian POWER
- IBM XL C compiler 13
  - all algorithms are implemented with VSX (128-bit SIMD instructions) using intrinsics
Throughput with and without pre-computation

- Higher throughput is better.
- Up to 75% improvements.
# Root Mean Squared Error

<table>
<thead>
<tr>
<th>Problem size</th>
<th>With pre-computation</th>
<th>Without pre-computation</th>
<th>Interpolation disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L=128$</td>
<td>0.534</td>
<td>0.513</td>
<td>12.088</td>
</tr>
<tr>
<td>$L=256$</td>
<td>0.538</td>
<td>0.517</td>
<td>12.108</td>
</tr>
<tr>
<td>$L=512$</td>
<td>0.538</td>
<td>0.518</td>
<td>12.118</td>
</tr>
<tr>
<td>$L=1024$</td>
<td>0.545</td>
<td>0.526</td>
<td>12.120</td>
</tr>
</tbody>
</table>

(lower is better)

- **Smiley face**: negligible degradation in image quality
- **Sad face**: significant degradation in image quality
## Overhead of Pre-Computation

<table>
<thead>
<tr>
<th>Problem size</th>
<th>With pre-computation</th>
<th>Without pre-computation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>precomputation</td>
<td>reconstruction</td>
</tr>
<tr>
<td>$L=128$</td>
<td>0.93 msec (47%)</td>
<td>1.11 msec</td>
</tr>
<tr>
<td>$L=256$</td>
<td>0.94 msec (22%)</td>
<td>3.36 msec</td>
</tr>
<tr>
<td>$L=512$</td>
<td>0.94 msec (4.1%)</td>
<td>22.20 msec</td>
</tr>
<tr>
<td>$L=1024$</td>
<td>0.93 msec (0.6%)</td>
<td>169.30 msec</td>
</tr>
</tbody>
</table>

- The numbers show the execution time per projection image.
- The percentages shown in parenthesis show the ratios to the total execution time.

- Average numbers of interpolations (i.e # voxles) per pixel
  - $L=128 \rightarrow 1.75$
  - $L=1024 \rightarrow 896$
Vector Unit Utilization

- **L=128**
- **L=256**
- **L=512**
- **L=1024**

- **with pre-computation**
- **without pre-computation**
- **interpolation disabled**

Higher is better.
System memory bandwidth requirements

The chart shows the total number of L3 cache misses for different problem sizes (L) with and without pre-computation, as well as with interpolation disabled. The lower the number, the better.

- **L=128**: With pre-computation, without pre-computation, and interpolation disabled.
- **L=256**: Similar categories as above.
- **L=512**: Similar categories as above.
- **L=1024**: Similar categories as above.

The chart indicates that pre-computation significantly reduces the number of cache misses compared to not using pre-computation, especially as the problem size increases.
Throughput with and without pre-computation using 3rd degree Lagrange interpolation

Throughput (GUPS)

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>With Pre-computation</th>
<th>Without Pre-computation</th>
<th>Interpolation Disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>L=128</td>
<td>3.5</td>
<td>2.5</td>
<td>1.5</td>
</tr>
<tr>
<td>L=256</td>
<td>3.0</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>L=512</td>
<td>2.5</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>L=1024</td>
<td>2.0</td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Higher is faster

Up to 57% improvements
Summary

- We developed an efficient way of interpolation from grid data at a non-grid point
  - Our pre-computation simplifies the computation drastically
  - The cost of pre-computation is not significant for realistic data size
- This technique is not specialized for CT reconstruction and applicable for other applications
- Refer the paper for more detail including:
  - Results for a more accurate interpolation algorithm
  - Performance modeling
  - Handling of floating point errors
  - NUMA optimization
backup
Scalability with Multiple NUMA Nodes

- **L=256**
  - 4.9x with 5 cores
  - 9.0x with 10 cores
  - 14.2x with 20 cores (1 NUMA node)

- **L=512**

- **L=1024**

Graph shows throughput (GUPS) on the y-axis and number of cores on the x-axis. Higher values indicate better performance.
Memory Optimization for NUMA Machine

- Each NUMA node processes a projection image independently from other NUMA nodes to avoid remote memory accesses
  - Within a NUMA node, all threads process one projection image by dividing voxels into small blocks
- We gather the partial results from each NUMA nodes after processing all projection images to sum up them
Scalability with Multiple NUMA Nodes

- **L=256**
  - with NUMA opt
  - without NUMA opt

- **L=512**
  - with NUMA opt
  - without NUMA opt

- **L=1024**
  - with NUMA opt
  - without NUMA opt

The graph shows the throughput (GUPS) on a y-axis against the number of cores on an x-axis. Each set of lines represents different grid sizes with and without NUMA optimization. The note indicates an increase of 18.9x with 20 cores.
Comparing to Previous RabbitCT Scores (L=512)

<table>
<thead>
<tr>
<th>Processor</th>
<th># Core / # Boards</th>
<th>Year</th>
<th>Source</th>
<th>GUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>POWER8 3.69 GHz</td>
<td>20 cores (2 sockets)</td>
<td>2015</td>
<td>Ours</td>
<td>20.5</td>
</tr>
<tr>
<td>POWER8 3.69 GHz</td>
<td>10 cores (1 socket)</td>
<td>2015</td>
<td>Ours</td>
<td>10.6</td>
</tr>
<tr>
<td>Westmere-EX 2.4 GHz</td>
<td>40 cores (4 sockets)</td>
<td>2011</td>
<td>Official ranking</td>
<td>8.3</td>
</tr>
<tr>
<td>nVidia GTX 670</td>
<td>2 boards</td>
<td>2014</td>
<td>Official ranking</td>
<td>152.9</td>
</tr>
</tbody>
</table>

- Today’s GPUs support bilinear interpolation in hardware!
- Our method will be beneficial even for GPUs when a higher-order interpolation algorithm is used