Fast Interpolation of Grid Data at a Non-Grid Point

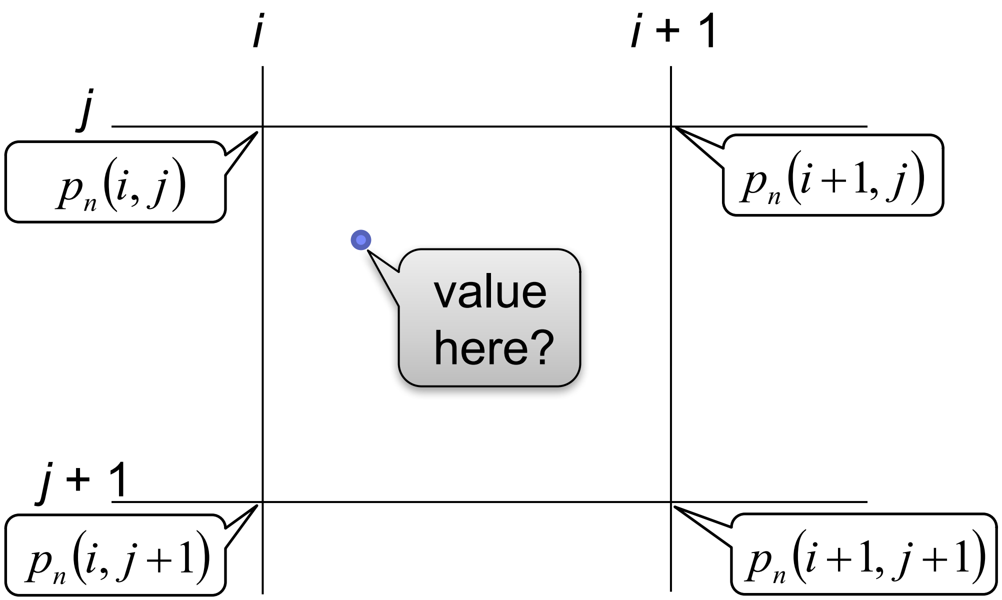
Hiroshi Inoue
IBM Research – Tokyo

December 12, 2017 | IEEE BigData 2017 @ Boston, MA, USA

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)

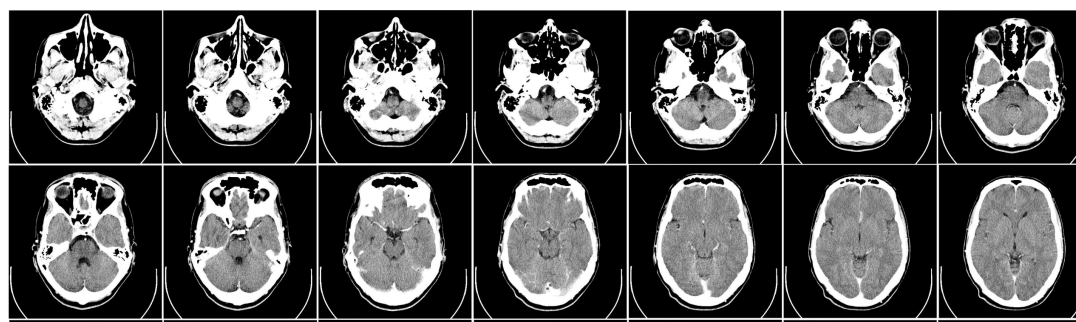
CT Reconstruction Overview

- 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

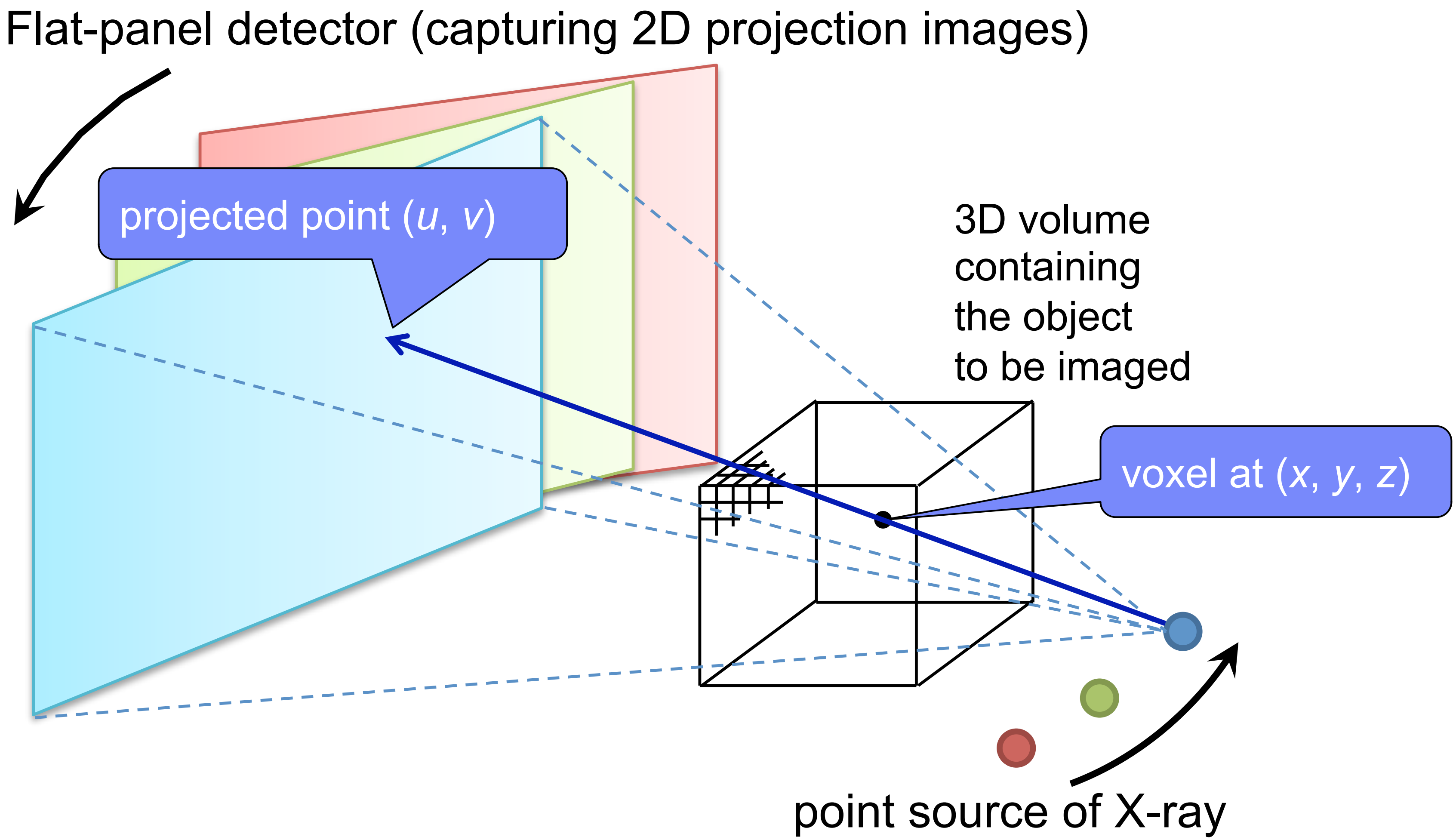
Source: http://www.sharpmedical.com/refurbished-c-arms/ziehm-c-arms/ziehm-exposcop-7000-c-arm/

Example of output from a CT system



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

Projection in CT Reconstruction



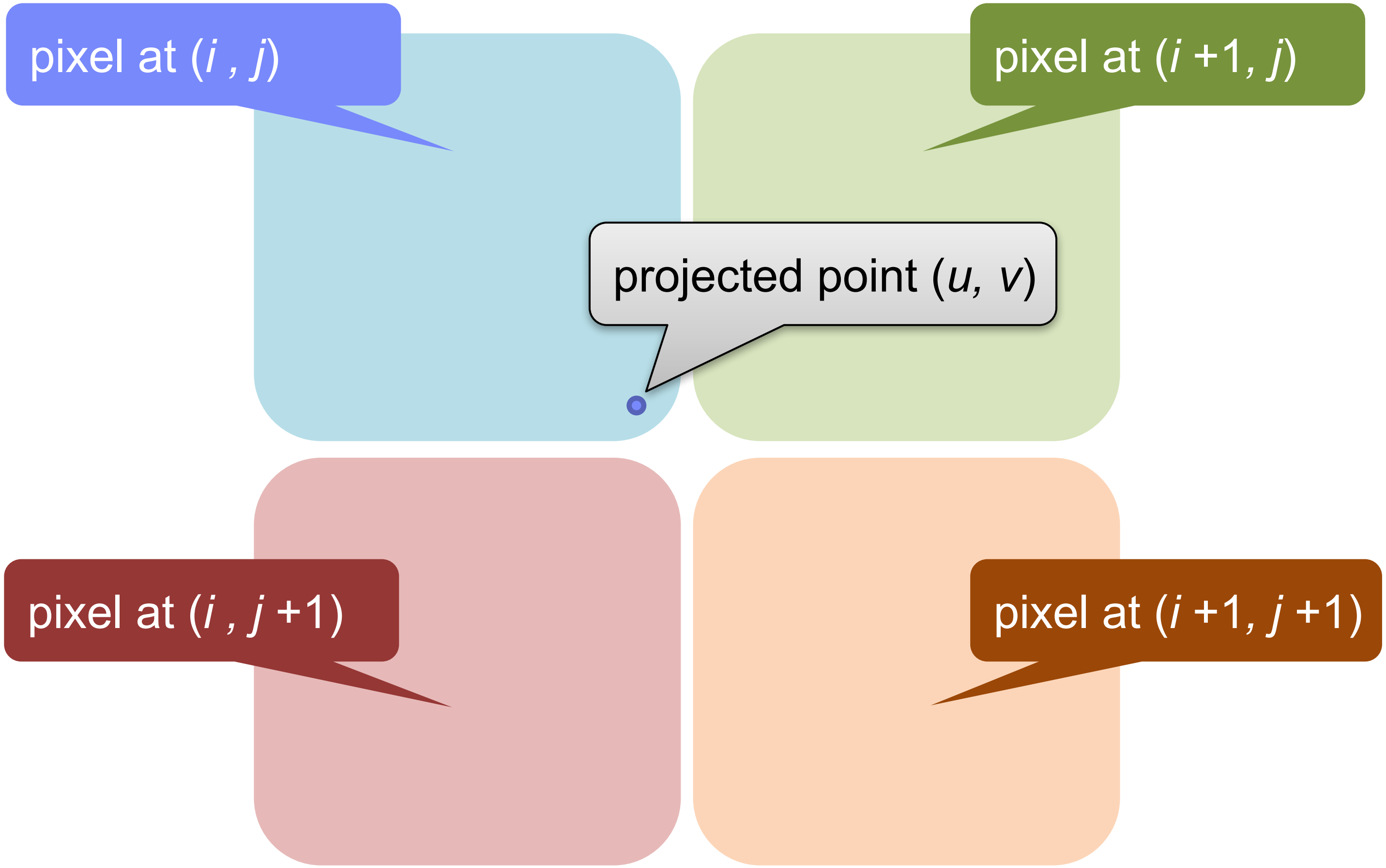
Baseline Reconstruction Algorithm Overview [2]

```
for each projection image In
  // reconstruction (projection)
  for z = 0 to L-1           ⎫
    for y = 0 to L-1         ⎬ for each voxel in 3D volume
      for x = 0 to L-1       ⎭
        1) project voxel (x,y,z) onto In
        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
```

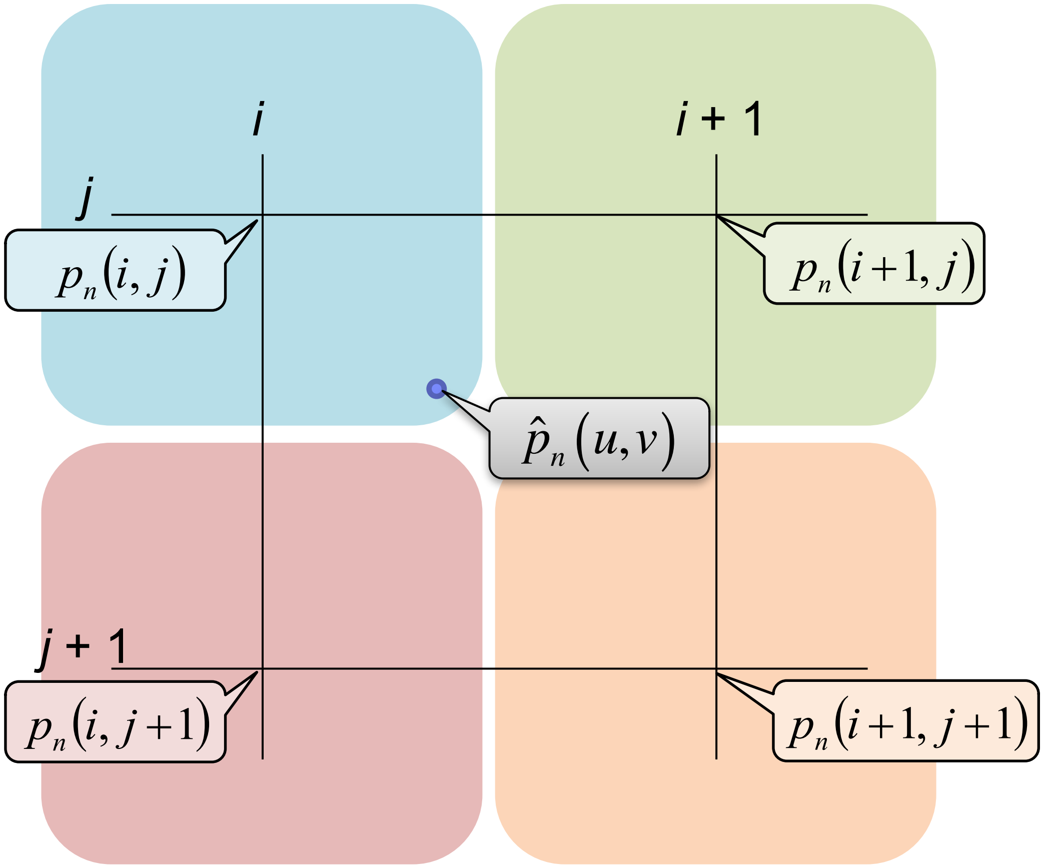
Baseline Reconstruction Algorithm Overview [2]

```
for each projection image I_n
  // reconstruction (projection)
  for z = 0 to L-1
    for y = 0 to L-1          } for each voxel in 3D volume
      for x = 0 to L-1
        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
```

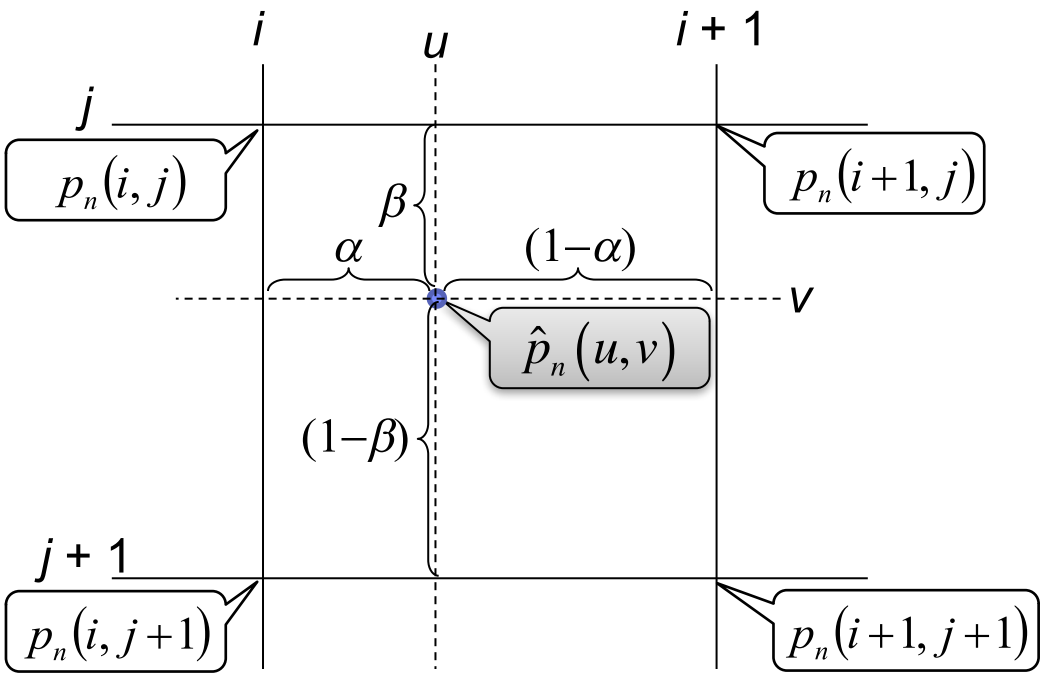
Interpolation from Grid Data at Non-Grid Point



Interpolation from Grid Data at Non-Grid Point



Interpolation from Grid Data at Non-Grid Point



$$\alpha = u - i = u - \lfloor u \rfloor$$

$$\beta = v - j = v - \lfloor v \rfloor$$

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?

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

☺ 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?

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

$$= (\mathbf{1} - \boldsymbol{\alpha} - \boldsymbol{\beta} + \boldsymbol{\alpha\beta}) \cdot p_n(i,j)$$
$$+ (\boldsymbol{\alpha} - \boldsymbol{\alpha\beta}) \cdot p_n(i+1,j)$$
$$+ (\boldsymbol{\beta} - \boldsymbol{\alpha\beta}) \cdot p_n(i,j+1)$$
$$+ \boldsymbol{\alpha\beta} \cdot p_n(i+1,j+1)$$

Group terms by the number of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$

$$= \boldsymbol{\alpha\beta} \cdot C_0(i,j) + \boldsymbol{\alpha} \cdot C_1(i,j) + \boldsymbol{\beta} \cdot C_2(i,j) + C_3(i,j)$$

$$C_0(i,j) = p_n(i,j) - p_n(i+1,j) - p_n(i,j+1) + p_n(i+1,j+1)$$

Our Efficient Interpolation Me

In the paper, we group terms based on $\boldsymbol{u}$ and $\boldsymbol{v}$ instead of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ for further performance boost

$$\alpha = u - i = u - \lfloor u \rfloor \quad \beta = v - j = v - \lfloor v \rfloor$$

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

$$= \alpha\big(\beta \cdot C_0(i,j) + C_1(i,j)\big) + \big(\beta \cdot C_2(i,j) + C_3(i,j)\big)$$

➔ If we have these four coefficients C_0 - C_3, we can compute this formula with only **<u>three multiply-and-add instructions!</u>**

C_0 - C_3 are independent from $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ (and hence *x*, *y*, *z*)
➔ We can pre-compute them at run time before iterating voxels and store in memory

<u>original (without pre-computation)</u>

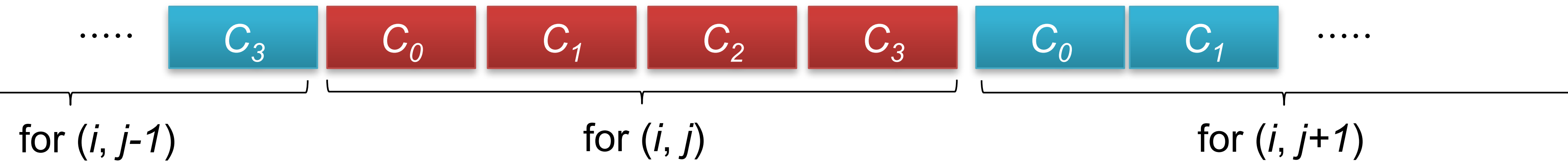
```
for each projection image
    projection
end
```

<u>with pre-computation</u>

```
for each projection image
    pre-computation
    projection
end
```

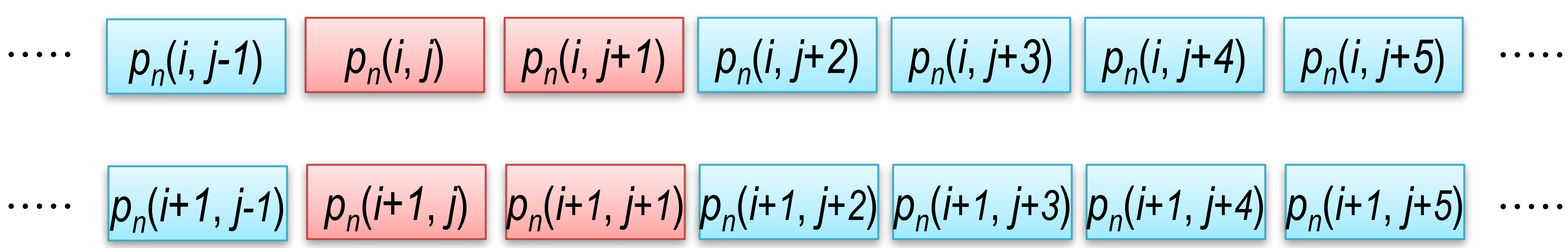
In-memory Pre-computed Table

Pre-computed Table



➔ ☹ 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



Overall Algorithm with Pre-Computation

```
for each projection image In
  // pre-computation
  for i = 0 to Sx-1
    for j = 0 to Sy-1
      calculate and store coefficients C0 to C3 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 In
        2) read coefficients C0 to C3 from pre-computed table
        3) estimate value at projected point by interpolation
        4) update density value of voxel (x,y,z)
      end
    end
  end
end
```

for each pixel in 2D projection image (loops over i, j)

for each voxel in 3D volume (loops over x, y, z)

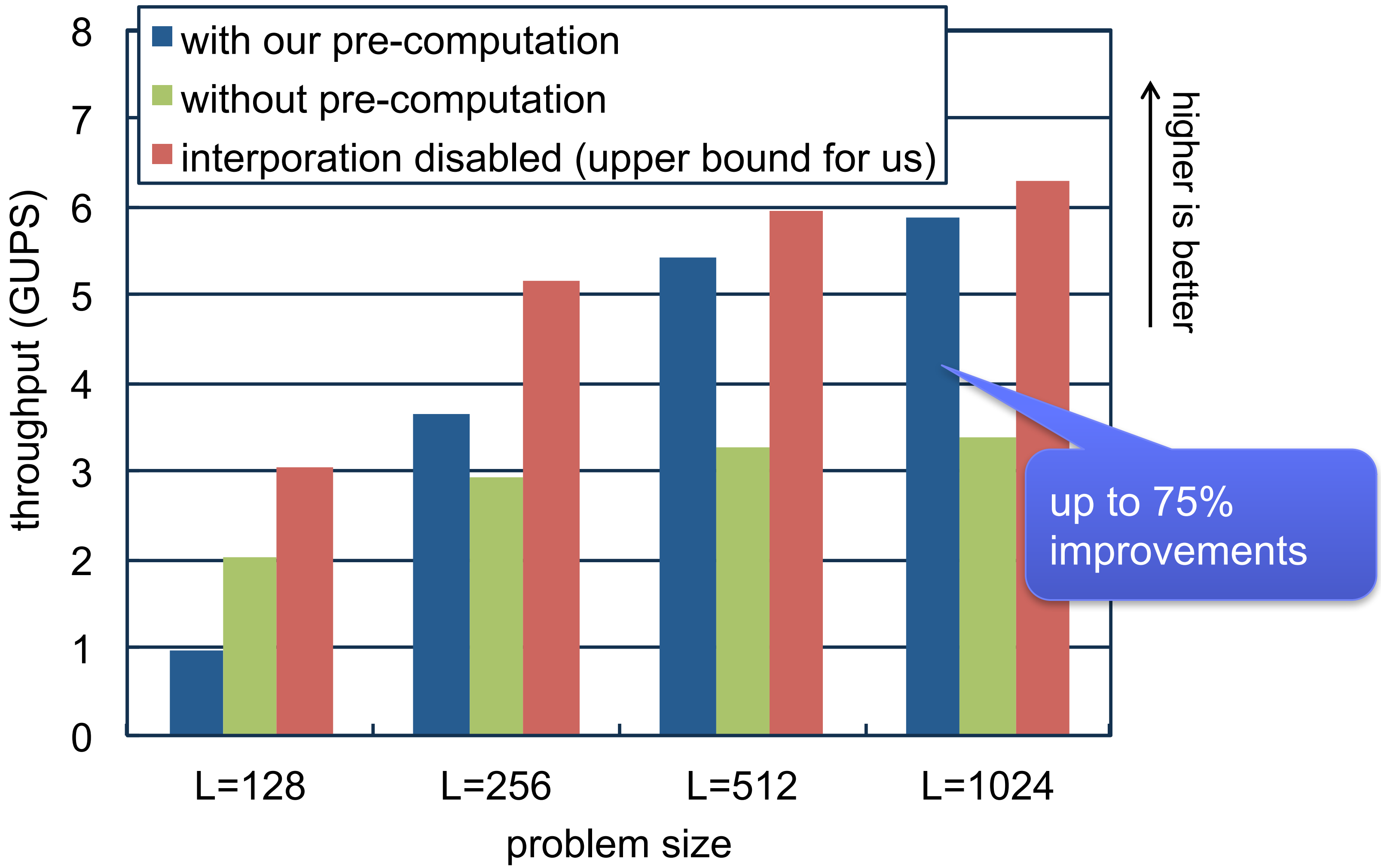
Performance Evaluation with RabbitCT

- 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



Root Mean Squared Error

(lower is better)

Problem size	With pre-computation	Without pre-computation	Interpolation disabled
L=128	0.534	0.513	12.088
L=256	0.538	0.517	12.108
L=512	0.538	0.518	12.118
L=1024	0.545	0.526	12.120

☺ negligible degradation in image quality

☹ significant degradation in image quality

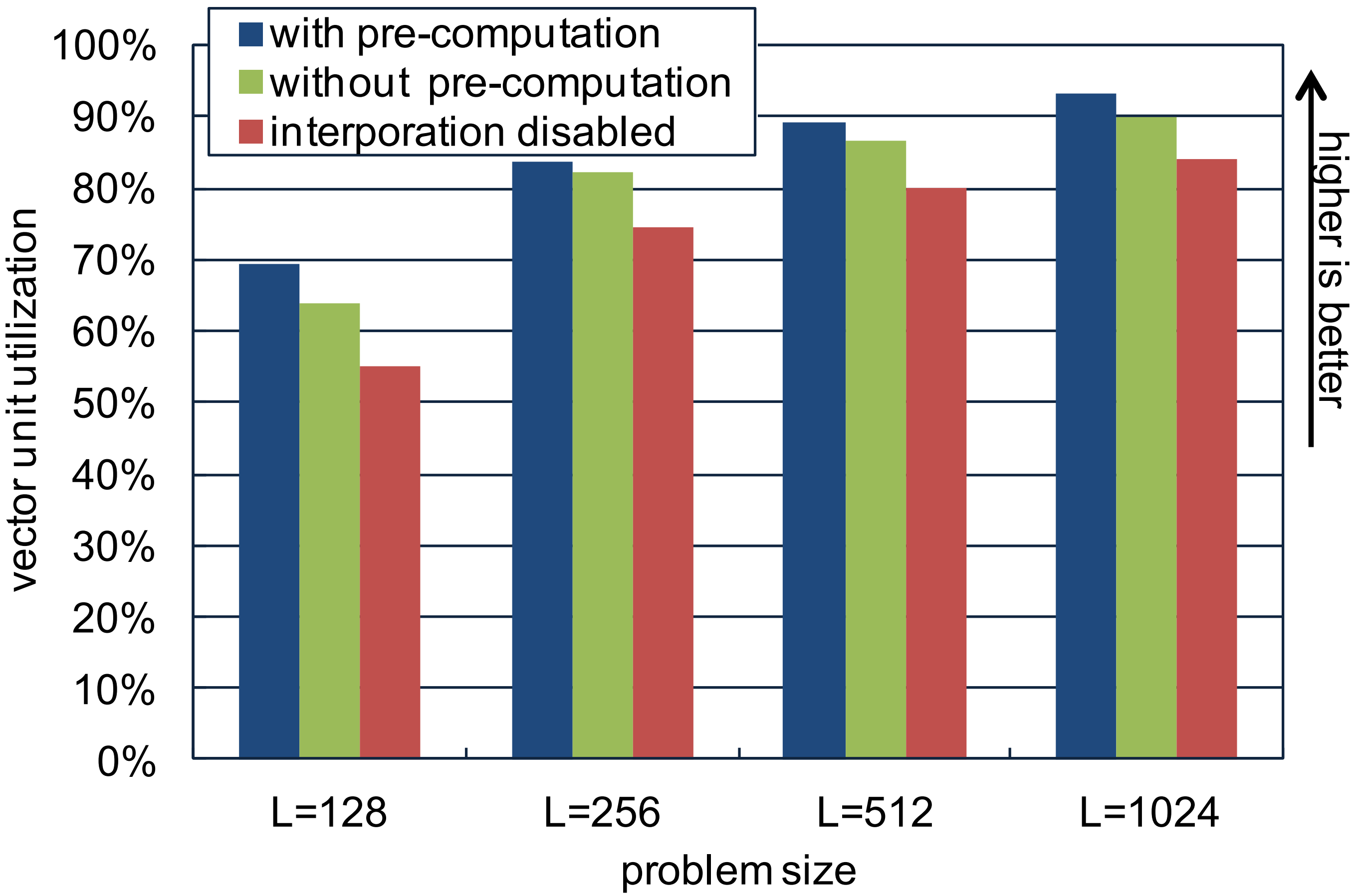
Overhead of Pre-Computation

Problem size	With pre-computation			Without pre-computation
	precomputation	reconstruction	total	
L=128	0.93 msec (47%)	1.11 msec	2.01 msec	0.94 msec
L=256	0.94 msec (22%)	3.36 msec	4.19 msec	5.21 msec
L=512	0.94 msec (4.1%)	22.20 msec	22.59 msec	37.90 msec
L=1024	0.93 msec (0.6%)	169.30 msec	167.02 msec	293.87 msec

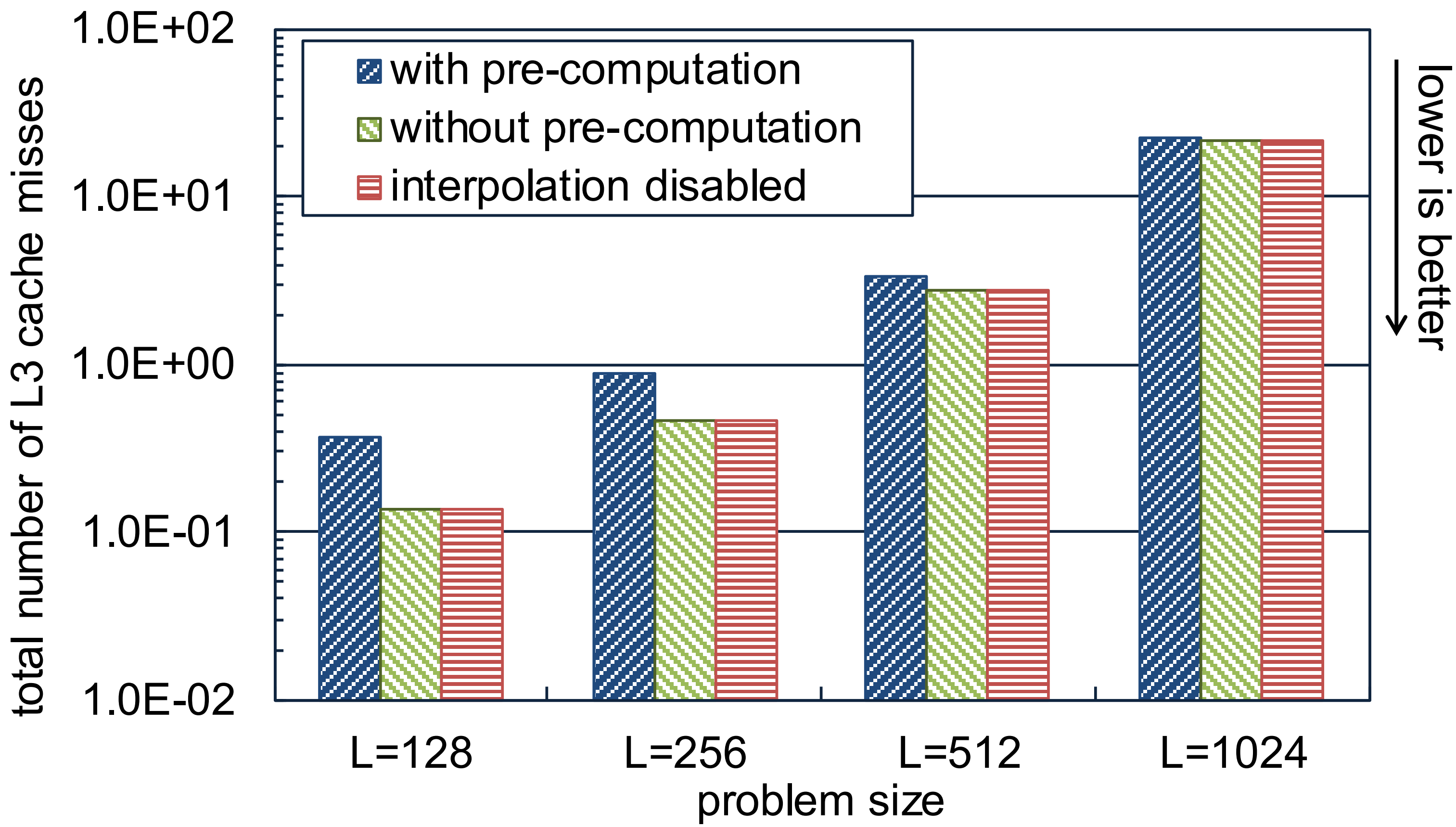
- The numbers show the execution time per projection image.
- The percentages shown in parenthesis show the ratios to the total execution time.

- Average numbers of interpolations (i.e # voxles) per pixel
  - L=128 → 1.75
  - L=1024 → 896

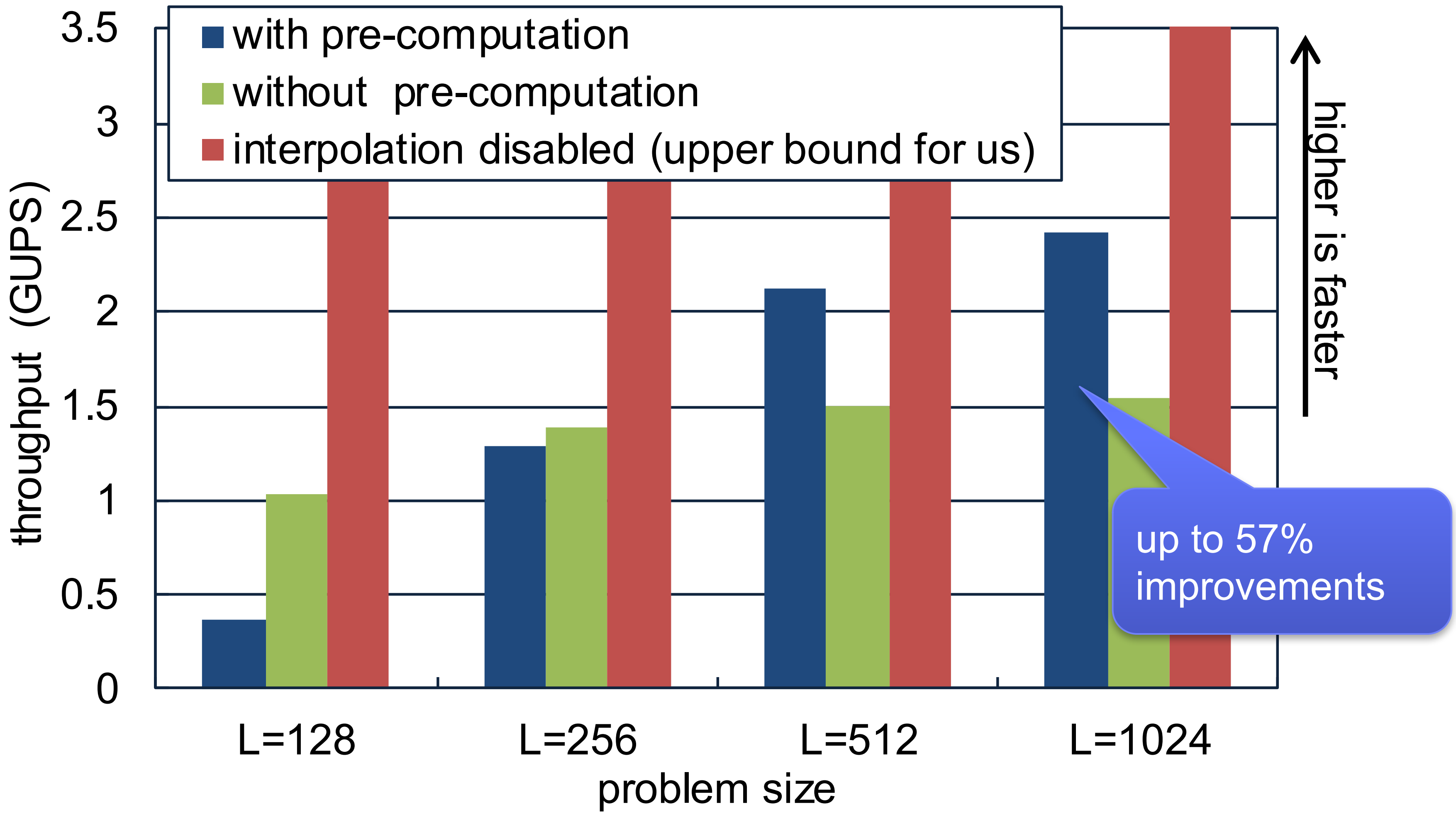
Vector Unit Utilization



System memory bandwidth requirements



Throughput with and without pre-computation using 3rd degree Lagrange interpolation



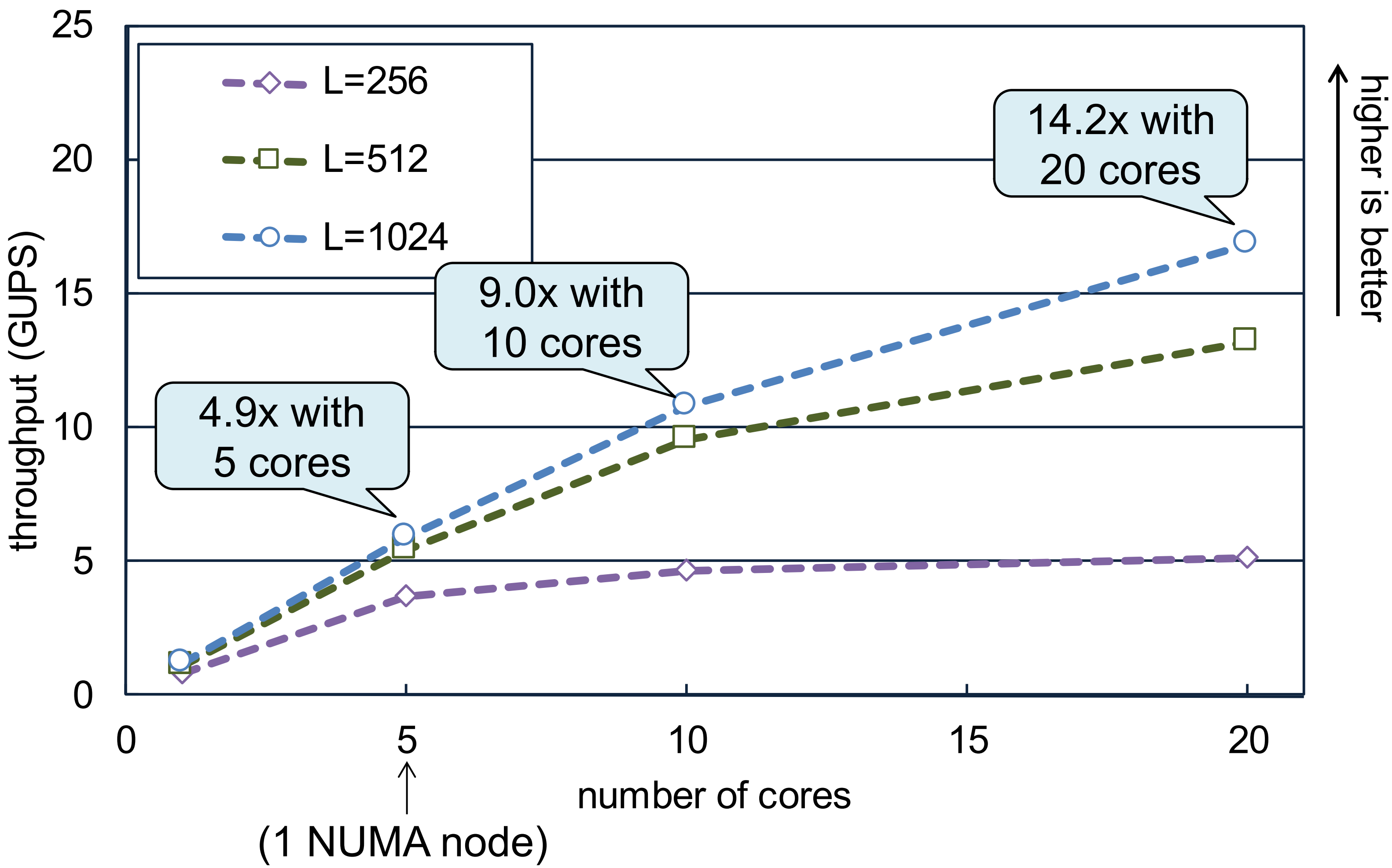
Summary

- 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

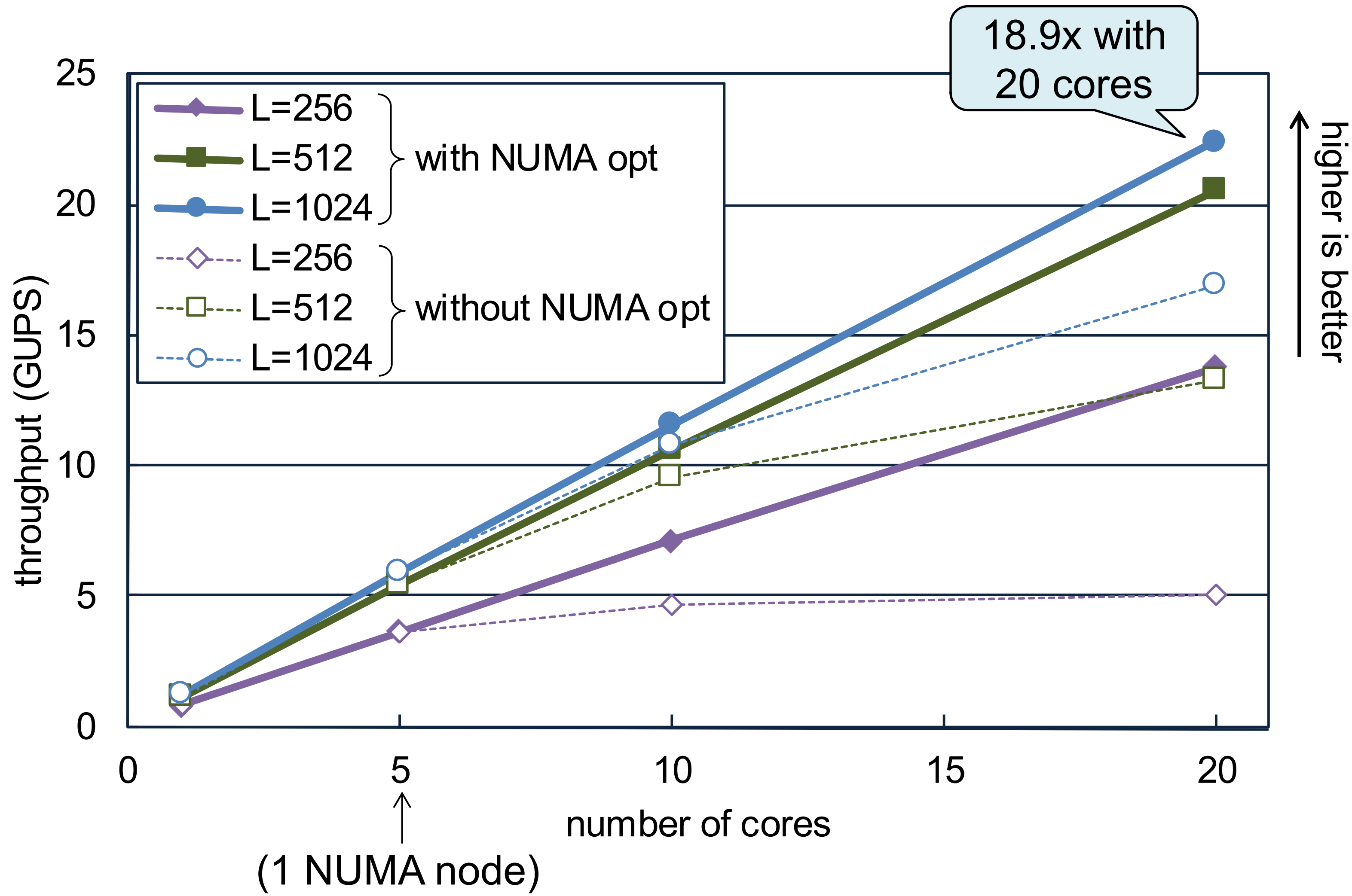
Scalability with Multiple NUMA Nodes



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



Comparing to Previous RabbitCT Scores (L=512)

Processor	# Core / # Boards	Year	Source	GUPS
POWER8 3.69 GHz	20 cores (2 sockets)	2015	Ours	20.5
POWER8 3.69 GHz	10 cores (1 socket)	2015	Ours	10.6
IvyBridge-EP 2.2 GHz	20 cores (2 sockets)	2014	Paper [4]	about 7.0
Westmere-EX 2.4 GHz	40 cores (4 sockets)	2011	Official ranking	8.3
Xeon Phi 5110P	1 board	2014	Paper [4]	about 8.5
nVidia GTX 670	2 boards	2014	Official ranking	152.9

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