Implementation of a Backprojection Algorithm on CELL

Mario Koerner

Moscow-Bavarian Joint Advanced Student School 2006
March 19 2006 to March 29 2006

Overview

- Practical implementation of 3D Backprojection
- Porting Backprojection to CELL
- Optimize backprojection for the CELL
  - Optimization on SPU level
  - Optimization of the data transfer
  - Optimization of subvolume scheduling

3D Backprojection

Reminder: the Feldkamp algorithm

Input: 2D Projection matrices and acquisition parameters

Step 1: Perform a proper weighting of projection images

Step 2: Filter the projection images along horizontal lines

Step 3: Perform a weighted backprojection along the projection rays

Initialize the reconstruction volume with zero

For all projections P_j , j=1...N
    For all voxels v_i , i=1...M

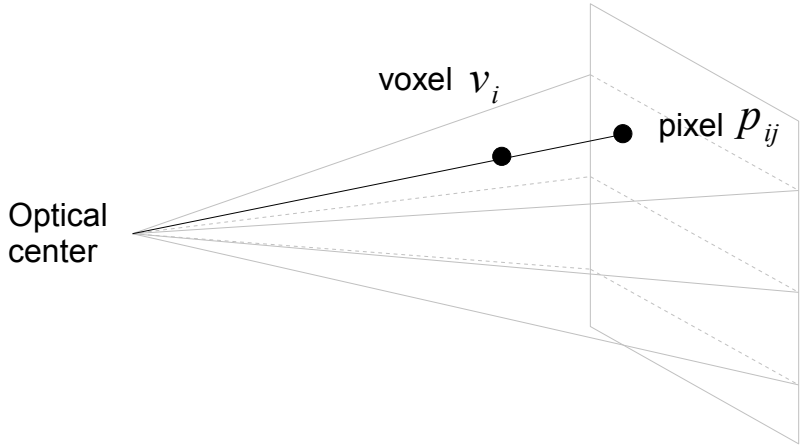
        Compute the coordinates p_{ij} of voxel v_i in projection P_j

        Get the projection value at this position

        Accumulate the weighted value to the reconstruction volume

Computation of projection coordinates

- The mapping between a voxel in the reconstruction volume and the corresponding pixel in the projection image is a central perspective projection



- It can be written as a linear mapping using homogeneous coordinates

$$\begin{pmatrix} r \\ s \\ t \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- The acquisition parameters are defined using this projection matrix

Computation of projection coordinates

- The columns of the projection matrix may be interpreted as incremental updates when navigating through the volume in x/y/z direction

$$\begin{pmatrix} r \\ s \\ t \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix} x + \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} y + \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \end{pmatrix} z + \begin{pmatrix} a_{14} \\ a_{24} \\ a_{34} \end{pmatrix}$$

- E.g. For navigation from voxel (x, y, z) to voxel (x, y+1, z), we have

$$\begin{pmatrix} r \\ s \\ t \end{pmatrix}' = \begin{pmatrix} r \\ s \\ t \end{pmatrix} + \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix}$$

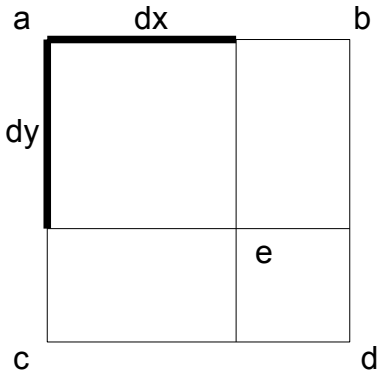
- The last column describes the „cube suspension“, i.e. the pixel, where voxel (0,0,0) is mapped to

- A division is required for computing the cartesian coordinates of the projection pixel

$$p_{ij} = (x_{ij}, y_{ij})^T = \left(\frac{r_{ij}}{t_{ij}}, \frac{s_{ij}}{t_{ij}}\right)$$

Retrieving projection values

- Because the projection images are measured as discrete functions, the computed coordinates may not correspond to a value in the data set.
- Use the „Nearest Neighbour“ approach
- Use bilinear interpolation



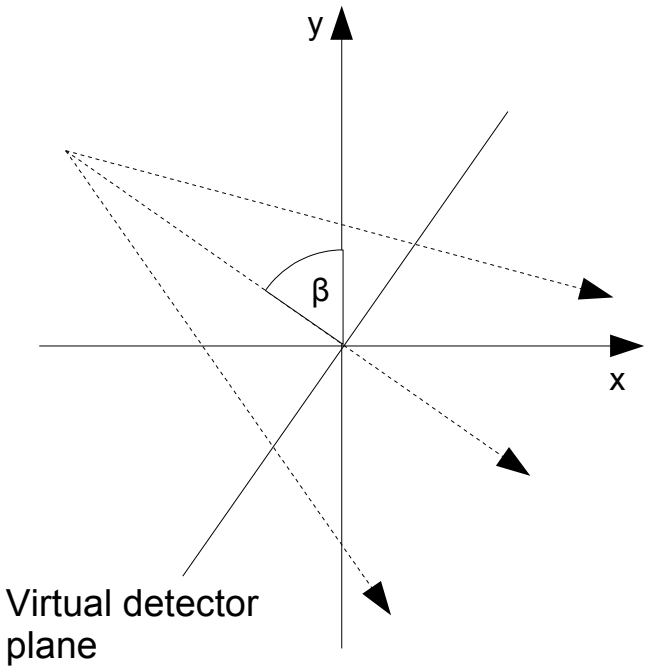
$$e=(1-dx)(1-dy)a+(1-dx)\,dy\,c+dx(1-dy)\,b+dx\,dy\,d$$

Computing weights for the BP

- In the Feldkamp algorithm, each voxel is weighted during the backprojection with

$$\frac{1}{U^2} \quad \text{with} \quad U(x,y,\beta)=\frac{D+x\sin\beta-y\cos\beta}{D}=\frac{\sin\beta}{D}x-\frac{\cos\beta}{D}y+1$$

- U can also be computed using incremental updates
- The increments can be found in the projection matrix



The fan can be rotated and shifted such that the central ray coincides with the y-axis and the source lies in the center of the coordinate system:

$$R=\begin{pmatrix}\cos(-\beta) & -\sin(-\beta) & 0 & 0\\ \sin(-\beta) & \cos(-\beta) & 0 & -D\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

Computing weights for the BP

- On the transformed fan position, we can apply the projection matrix

$$P = \begin{pmatrix} D & 0 & 0 & 0 \\ 0 & 0 & D & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix}$$

- The multiplication of both matrices gives

$$P\,R = \begin{pmatrix} D\cos\beta & D\sin\beta & 0 & 0 \\ 0 & 0 & D & 0 \\ \sin\beta & -\cos\beta & 0 & D \end{pmatrix}$$

- and can be normalized to

$$\frac{P\,R}{D} = \begin{pmatrix} \cos\beta & \sin\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{\sin\beta}{D} & \frac{-\cos\beta}{D} & 0 & 1 \end{pmatrix}$$

- So the weighting factor U for the backprojection is equal to the third homogeneous coordinate t. This saves one division for the backprojection.

Overview

- Practical implementation of 3D Backprojection
- Porting Backprojection to CELL
- Optimize backprojection for the CELL
  - Optimization on SPU level
  - Optimization of the data transfer
  - Optimization of subvolume scheduling

Porting Backprojection to CELL

- The Backprojection algorithm needs to handle huge amounts of data:

  Volume: 512 x 512 x 512 voxels, 32 bit floating point numbers
  --> 512 MB

  Projections: 500 Projections, 1024 x 1024 Pixels, 32 bit floating point numbers
  --> 2 GB

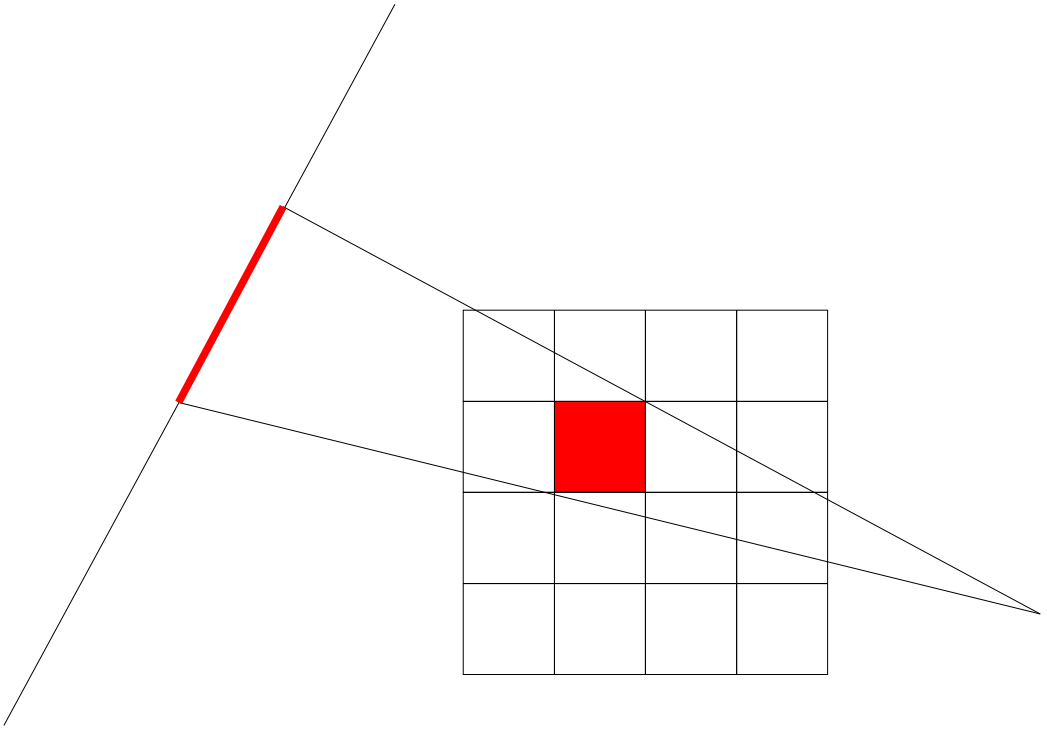
- The data must be partitioned into chunks that fit into the 256K local store of the SPUs

- Data Partitioning for the backprojection algorithm is easy, since all voxels in the reconstructed volume and all projections can be considered independently.

- Because we sample in the space of the output by projecting voxels into projection images, it is intuitive to split the volume first and compute the areas of the projections required by these subvolumes.

Projection shadows

- To compute the increments for a subvolume from a specific projection, only a small sector of the projection image is required



- This can be computed by projecting all 8 corners of the subvolume into the image plane and getting the bounding box of them

- The lines of the image section must be aligned by 16 bytes (4 pixels) and have a length of a multiple of 16 bytes, so that they can be transfered by the DMA controller.

Work partitioning between PPU and SPUs

- Basic reconstruction task consists of:
  - A section of the reconstruction volume
  - The shadow of this volume in one projection image
- In a PPU centric implementation, the PPU is responsible for
  - Performing all I/O operations for loading and storing data
  - Do the scheduling (when a task is to be computed)
  - Do the placing (where a task is to be computed)
  - Avoid conflicts (2 SPUs writing to the same subvolume at the same time)
  - Compute the projection shadows
- The SPUs
  - Load the data required for the basic reconstruction task
  - Perform the actual backprojection
  - Write the results back to main memory
- Basic reconstruction tasks are posted to the SPUs through their mailbox registers

First results

- The code generated by gcc for the backprojection function is highly inefficient:

```
Single cycle                                          200428 ( 34.4%)
Dual cycle                                             50813 (  8.7%)
Nop cycle                                               4098 (  0.7%)
Stall due to branch miss                               13396 (  2.3%)
Stall due to prefetch miss                                 0 (  0.0%)
Stall due to dependency                               313989 ( 53.8%)
Stall due to fp resource conflict                          0 (  0.0%)
Stall due to waiting for hint target                     567 (  0.1%)
Stall due to dp pipeline                                   0 (  0.0%)
Channel stall cycle                                        0 (  0.0%)
SPU Initialization cycle                                   0 (  0.0%)
-------------------------------------------------------------------
Total cycle                                           583291 (100.0%)
```

- The computation time for a 512 x 512 x 512 cube and 1 projection would be (on 1 SPU with 2.1 GHz)

$$T = \frac{512^3}{16^3} \frac{583291\,cycles}{2.1\,GHz} = 9.1\,s$$

Overview

- Practical implementation of 3D Backprojection
- Porting Backprojection to CELL
- Optimize backprojection for the CELL
  - Optimization on SPU level
  - Optimization of the data transfer
  - Optimization of subvolume scheduling

The inner loop of the BP

Initialize the reconstruction volume with zero

For all projections P_j , j=1...N
For all voxels v_i , i=1...M

Compute the coordinates p_{ij} of voxel v_i in projection P_j

Get the projection value at this position

Accumulate the weighted value to the reconstruction volume

- Compute the homogeneous coordinates by adding the offset in x-direction

```
r += drx;
s += dsx;
t += dst;
```

3 adds per voxel (float)

- Compute the cartesian coordinates by dehomogenisation

```
one_over_t = 1 / t;
x = r * one_over_t;
y = s * one_over_t;
```

1 division, 2 multiplications per voxel (float)

The inner loop of the BP

Initialize the reconstruction volume with zero

For all projections P_j , j=1...N
For all voxels v_i , i=1...M

Compute the coordinates p_{ij} of voxel v_i in projection P_j

Get the projection value at this position

Accumulate the weighted value to the reconstruction volume

- Compute the index of the voxel within the subimage

```
Index = SubSizeX * (x - FirstSubPixelX)
        + (y - FirstSubPixelY);
```

2 subs, 1 mult-add per voxel (integer)

- Load the projection value from local store

```
Value = projection[index];
```

1 memory access at a 4 byte aligned location

The inner loop of the BP

Initialize the reconstruction volume with zero

For all projections P_j , j=1...N
For all voxels v_i , i=1...M

Compute the coordinates p_{ij} of voxel v_i in projection P_j

Get the projection value at this position

Accumulate the weighted value to the reconstruction volume

- Compute weight for this pixel

```
W = one_over_t * one_over_t;
```

1 multiplication per voxel (float)

- Load the projection value from local store

```
Volume[pos++] += W * Value;
```

1 multiply-add (float)
1 add (integer)

Vectorization

Initialize the reconstruction volume with zero

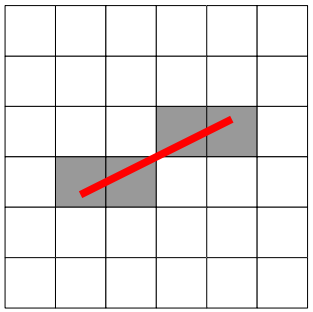
For all projections P_j , j=1...N
For all voxels v_i , i=1...M

Compute the coordinates p_{ij} of voxel v_i in projection P_j

Get the projection value at this position

Accumulate the weighted value to the reconstruction volume

- Steps 1 and 3 can be vectorized by applying all operations to 4 voxels in parallel
- The memory access in step 2 cannot be vectorized, because the required pixels may not lie in the same row



It should be possible for the compiler to schedule the required load and shuffle instructions on the odd pipeline, while arithmetic operations are running on the even pipeline.

Required operations per voxel

- Floating point arithmetics:
    - 3 additions
    - 1 division
    - 3 multiplications
    - 1 multiply-add
- Integer arithmetics:
    - 3 addtions
    - 1 multiply-add
- Problem: SPUs do not support exact division in hardware
    - Hardware support for computing an estimate for the reciprocal of a floating point number
    - A SDK library function exists for performing IEEE compliant divisions
    - A fast version of the library function exists that gives 32 bit float precision

Reciprocal estimate in hardware

- Uses the `frest` (floating reciprocal estimate) and `fi` (floating interpolate) instructions that are combined in the `spu_re()` intrinsic
- The resulting precision is 12 bit
- Result for performing 4 vector divisions:

```
Single cycle                                9 ( 56.2%)
Dual cycle                                  4 ( 25.0%)
Nop cycle                                   0 (  0.0%)
Stall due to branch miss                    0 (  0.0%)
Stall due to prefetch miss                  0 (  0.0%)
Stall due to dependency                     3 ( 18.8%)
Stall due to fp resource conflict           0 (  0.0%)
Stall due to waiting for hint target        0 (  0.0%)
Stall due to dp pipeline                    0 (  0.0%)
Channel stall cycle                         0 (  0.0%)
SPU Initialization cycle                    0 (  0.0%)
-----------------------------------------------------
Total cycle                                16 (100.0%)
```

(includes a multiplication in order to perform a real division)

IEEE division

- Uses the library function `_divide_v()` with the define `IEEE_ACCURATE_DIVIDE`
- Does a lot of checks and sets several bits from the IEEE specification (e.g. overflows)
- Result for performing 4 vector divisions:

```
Single cycle                              112 ( 93.3%)
Dual cycle                                  7 (  5.8%)
Nop cycle                                   0 (  0.0%)
Stall due to branch miss                    0 (  0.0%)
Stall due to prefetch miss                  0 (  0.0%)
Stall due to dependency                     1 (  0.8%)
Stall due to fp resource conflict           0 (  0.0%)
Stall due to waiting for hint target        0 (  0.0%)
Stall due to dp pipeline                    0 (  0.0%)
Channel stall cycle                         0 (  0.0%)
SPU Initialization cycle                    0 (  0.0%)
-----------------------------------------------------
Total cycle                               120 (100.0%)
```

Fast division

- Uses the library function `_divide_v()` without the define `IEEE_ACCURATE_DIVIDE`
- Adds 1 Newton iteration after performing the reciprocal estimate in hardware to increase the accuracy
- Result for performing 4 vector divisions:

```
Single cycle                              37 ( 77.1%)
Dual cycle                                 0 (  0.0%)
Nop cycle                                  0 (  0.0%)
Stall due to branch miss                   0 (  0.0%)
Stall due to prefetch miss                 0 (  0.0%)
Stall due to dependency                   11 ( 22.9%)
Stall due to fp resource conflict          0 (  0.0%)
Stall due to waiting for hint target       0 (  0.0%)
Stall due to dp pipeline                   0 (  0.0%)
Channel stall cycle                        0 (  0.0%)
SPU Initialization cycle                   0 (  0.0%)
-----------------------------------------------------
Total cycle                               48 (100.0%)
```

Fast division – extra operations

- Using the fast division function from the SDK library requires the following additional operations:
  - `frest` (floating reciprocal estimate) and `fi` (floating interpolate)
  - 1 multiplication
  - 2 negative vector multiply and subtract
  - 1 multiply-add
  - 1 add
  - 1 vector compare
  - 1 selection instruction (bitwise)

First optimization results

- Using vector instructions for the computational expensive parts, the SPU statistics are as follows:

```
Single cycle                                53665 ( 35.3%)
Dual cycle                                  12658 (  8.3%)
Nop cycle                                    2834 (  1.9%)
Stall due to branch miss                    14147 (  9.3%)
Stall due to prefetch miss                      3 (  0.0%)
Stall due to dependency                     66568 ( 43.7%)
Stall due to fp resource conflict               0 (  0.0%)
Stall due to waiting for hint target         2312 (  1.5%)
Stall due to dp pipeline                        0 (  0.0%)
Channel stall cycle                             0 (  0.0%)
SPU Initialization cycle                        0 (  0.0%)
-----------------------------------------------------------
Total cycle                                152187 (100.0%)
```

- Speedup compared to the scalar version

$$Speedup = \frac{583291}{152187} = 3.83$$

$$T = \frac{512^3}{16^3} \frac{152187\,cycles}{2.1\,GHz} = 2.4\,s$$

First optimization results

- Using vector instructions for the computational expensive parts, the SPU statistics are as follows:

```
Single cycle                                53665 ( 35.3%)
Dual cycle                                  12658 (  8.3%)
Nop cycle                                    2834 (  1.9%)
Stall due to branch miss                    14147 (  9.3%)
Stall due to prefetch miss                      3 (  0.0%)
Stall due to dependency                     66568 ( 43.7%)
Stall due to fp resource conflict               0 (  0.0
Stall due to waiting for hint targ           2312 (  1.5
Stall due to dp                                 0 (  0.0
Channel stall c                                 0 (  0.0
SPU Initializat                                 0 (  0.0
----------------                            -------------
Total cycle                                152187 (100.0
```

The dual issue rate is still very low

It should be possible to reduce dependency stalls using loop unrolling.

Transforming the loops could improve branch hint efficiency.

- Speedup compared to the scalar version

$$Speedup = \frac{583291}{152187} = 3.83$$

$$T = \frac{512^3}{16^3} \frac{152187\,cycles}{2.1\,GHz} = 2.4\,s$$

Conclusions

- The overall performance will be dominated by the computation time on the SPUs
- If the computation can be optimized by another factor of 5, we can backproject about 30 projections per second on a 512 x 512 x 512 volume using 16 SPUs:

$$\frac{2.4\,s}{16 \cdot 5} = \frac{1}{33.3}\,s$$

- The amount of data to be transferred in 1 second then is

$$x \cdot 2 \cdot 512\,MB + y \cdot 30 \cdot 4\,MB$$

- E.g. if the volume data is loaded only once and each projection pixel must be loaded 32 times

$$1 \cdot 2 \cdot 512\,MB + 32 \cdot 30 \cdot 4\,MB = 4864\,MB$$

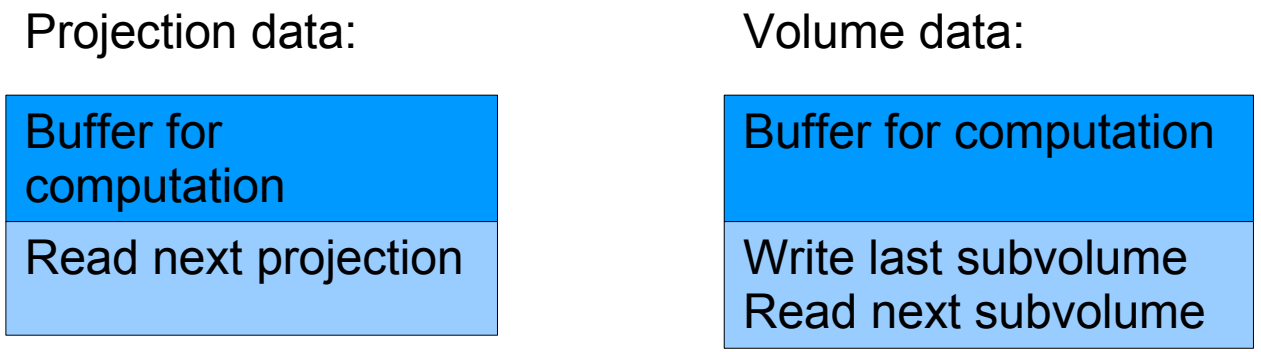
- Optimization of the SPU backprojection code and the code required to control the backprojection should have the highest priority for now.

Overview

- Practical implementation of 3D Backprojection
- Porting Backprojection to CELL
- Optimize backprojection for the CELL
  - Optimization on SPU level
  - Optimization of the data transfer
  - Optimization of subvolume scheduling

Optimization of data transfer

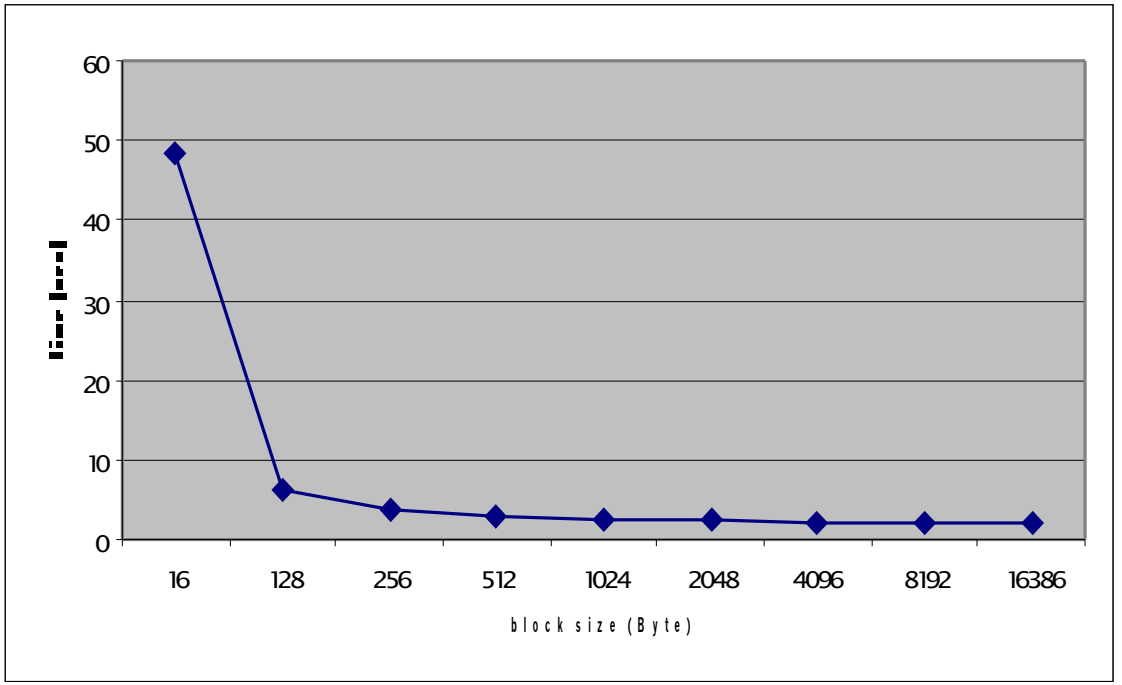
- The DMA controllers of the MFCs on the SPEs allow overlapping of computation with communication using the 'double buffering' technique
- Projection data can be discarded after it was backprojected into the reconstruction volume
- Volume data has to be written back to main memory after the computation
- Two buffers are sufficient for the volume data, if the 'fenced' version of the get command is used



- When using DMA lists for the transfer of volume data, at least 3 buffers for these DMA lists are required

Optimization of data transfer

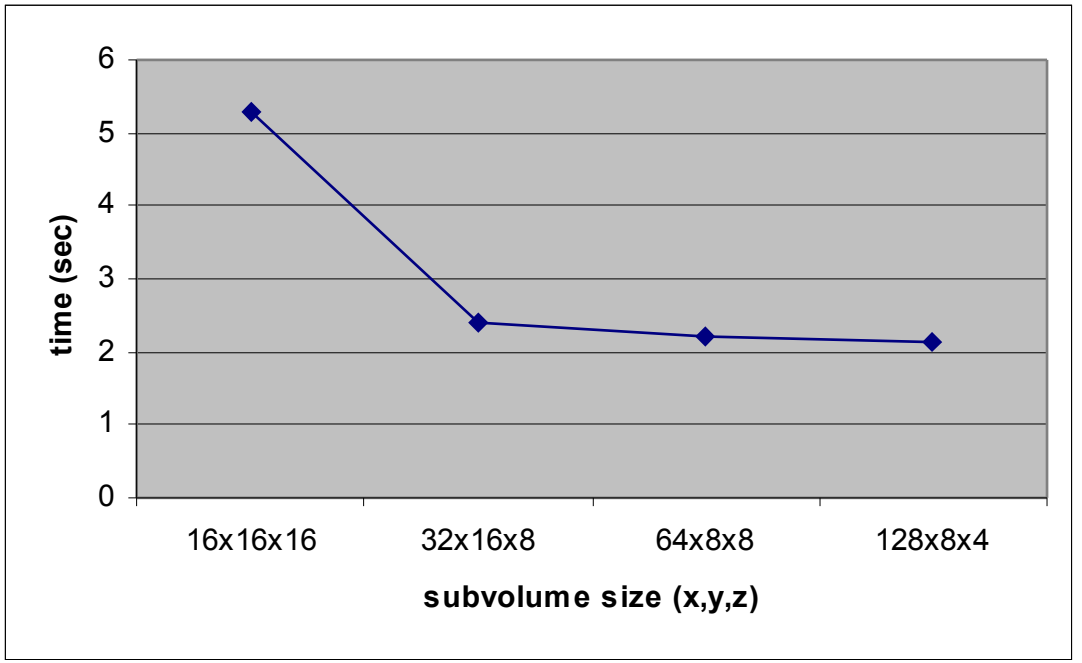
- Transfer of small memory segments is very inefficient



- Transfer of 50 GB of data
- Block size is the amount of data for one DMA
- This experiment used direct DMA commands (no DMA lists)
- Data rate: 22.7 GB per second

Optimization of data transfer

- This also holds for the transfer of subvolumes that are stored line by line



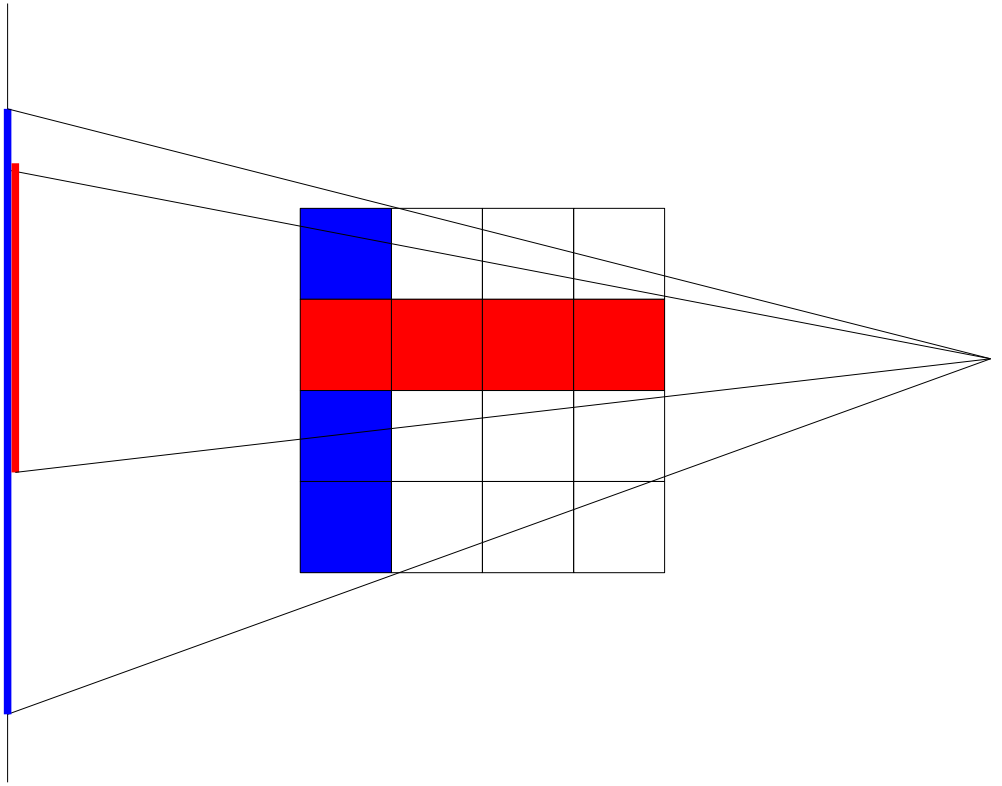
- This gives restrictions to the block shape that can be used for volume partitioning
- It may be better to store the volume in small blocks instead of lines and reorganize the data layout at the end of the computation

Overview

- Practical implementation of 3D Backprojection
- Porting Backprojection to CELL
- Optimize backprojection for the CELL
  - Optimization on SPU level
  - Optimization of the data transfer
  - Optimization of subvolume scheduling

Projection shadows

- The size of the projection shadow depends on the shape of the subvolume



- The optimal subvolume shape also depends on the projection angle

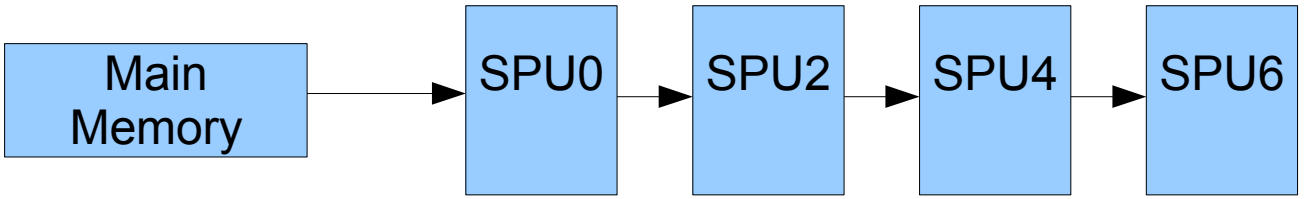
Subvolume scheduling

- An optimal solution would
  - Schedule (when?)
  - Bind (where?)

  the backprojection tasks in a way, that overall computation time is minimal
- If all tasks are approximately of the same size, load balancing is easy
- Therefore it should be sufficient to find an ordering that minimizes data transfers.

Pipelining

- If access to main memory becomes a bottleneck, data should held within the cell chip as long as possible
- One way for doing so: establish a pipeline from SPU to SPU and send projection data from stage to stage



- This requires distribution of a subvolume over the SPUs. The shadow of the complete subvolume must be small enough to fit into the spu.

References

[CBEA] Cell Broadband Engine Architecture. IBM Corporation, 2005

[CBETut] Cell Broadband Engine Programming Tutorial. IBM Corporation, 2005

[Kak] A. C. Kak and Malcolm Slaney, Principles of Computerized Tomographic Imaging. Society of Industrial and Applied Mathematics, 2001

[Tur] Henrik Turbell, Cone-Beam Reconstruction using Filtered Backprojection. Dissertation, Linköping Studies in Science and Technology, 2001