

Motivation

- Image registration is a key method in medical image analysis.
- Semi-Lagrangian solver for linear advection crucial component of the involved numerical algorithms.
- Interpolation is one of the two major compute kernels.

Constrained Diffeomorphic Image Registration

Problem Statement Find a plausible map $\mathbf{y}$ such that the *template image* m_T becomes similar to the reference image m_R , i.e. $m_R \approx m_T \circ \mathbf{y}$

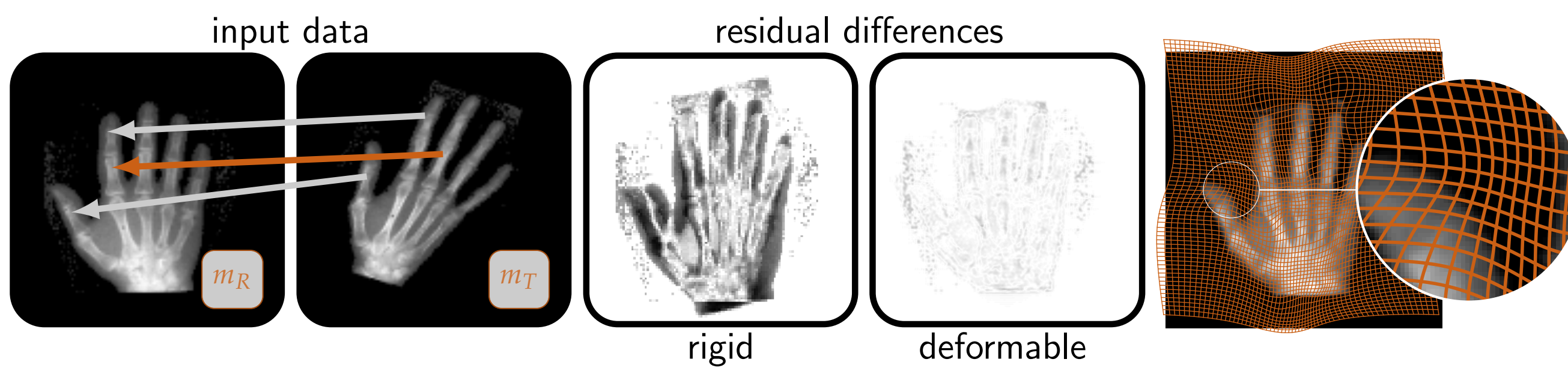


Figure 1: Illustration of an image registration problem. From left to right: Original images (reference image m_R and template image m_T (image to be registered to m_R)); transformed template image $m_T \circ \mathbf{y}$ and residual differences after rigid registration ($\mathbf{y}$ models rotation and translation); residual differences and deformed template image $m_T \circ \mathbf{y}$ after deformable registration. We require $\mathbf{y}$ to be a diffeomorphic map to render it physically meaningful.

Mathematical Formulation

We consider image registration as an inverse transport problem, which results in a partial differential equation (PDE) constrained optimization problem. The PDE constraints are the transport equations for the image intensities.

Find the velocity $\mathbf{v}$ of the deformation map $\mathbf{y}$ such that

$$\min_{m, \mathbf{v}} \text{distance}(m_R, m_1) + \frac{\beta}{2} \text{regularization}(\mathbf{v}) \quad \text{subject to} \quad \partial_t m + \mathbf{v} \cdot \nabla m = 0, \quad m_0 = m_T \quad (1)$$

additional (bio-physical) constraints on $\mathbf{v}$

Semi-Lagrangian Time Stepping Scheme

- Transport equation (1) is solved using semi-Lagrangian scheme that requires costly interpolation of velocities and image data along backward characteristics.
- A single image registration problem requires solving equation (1) multiple times for different velocity fields, creating a bottleneck for performance.

Algorithm 1: Semi-lagrangian Time Stepping Algorithm

Input: velocity field $\mathbf{v}(\mathbf{x})$, scalar field $m(\mathbf{x}, t = 0)$

Output: advected scalar field $m(\mathbf{x}, t = 1)$

```

1:  $\delta t \leftarrow \frac{1}{n_t}$ 
2:  $\mathbf{X}_* = \mathbf{x} - \delta t \mathbf{v}(\mathbf{x})$ 
3:  $\mathbf{X} = \mathbf{x} - \frac{\delta t}{2} (\mathbf{v}(\mathbf{x}) + \text{interp3D}(\mathbf{v}, \mathbf{X}_*))$ 
4: for  $k = 1 \rightarrow n_t$  do
5:    $m(\mathbf{x}, t = k\delta t) = \text{interp3D}(m(\mathbf{x}, t = (k-1)\delta t), \mathbf{X})$ 
6: end for
```

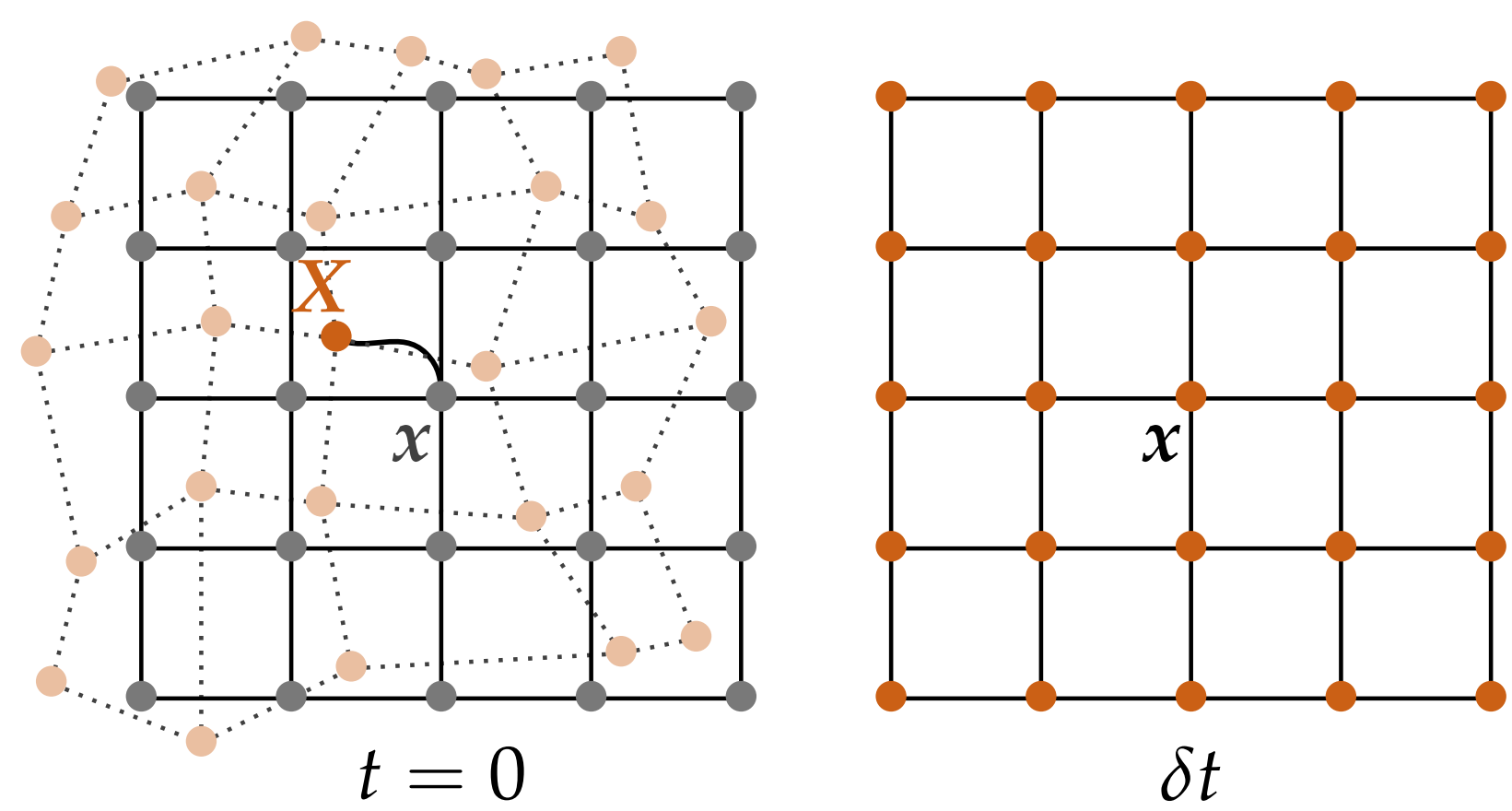


Figure 2: Illustration of the semi-Lagrangian scheme in 2-dimensions. Point $\mathbf{x}$ is a regular grid point and $\mathbf{X}$ is the material point at $t = 0$ that landed at $\mathbf{x}$ at time t . The semi-Lagrangian algorithm requires interpolation of $\mathbf{v}$ and m at these off-grid points

Computational Challenges

- Algorithmic, computational, and mathematical challenges (computational complexity; vast number of unknowns; instabilities; uniqueness of a solution...)
- Inverse problem solved via a globalized, preconditioned, Gauss–Newton–Krylov method which requires solving the transport equation multiple times at each optimization step.

GPU Accelerated SemiLagrangian Solver

Approach

- Original idea to perform cubic spline interpolation by doing a series of weighted linear interpolations was introduced by [1, 2, 3, 4]. The linear interpolation component in their kernel is accelerated by utilizing the texture filtering feature of modern GPUs.
- Computation of spline coefficients (prefiltering) is needed for greater interpolation accuracy and unless implemented carefully, is computationally as expensive as interpolation.
- Lagrange polynomials do not need prefiltering because they pass through the data points exactly.
- We apply the ideas presented in [1, 2, 3, 4] to perform interpolation on a GPU using Lagrange polynomials. Algorithm 2 describes our proposed method and figure 3 provides a visual of the computational stencil.

Algorithm Complexity

- The overall algorithmic complexity of the SemiLagrangian time stepping scheme is $d + n_t$ where d is the dimensionality of the problem and n_t is the number of discrete time steps.
- The scheme takes ~ 30 -50% of overall image registration costs.

Algorithm 2: 3D Lagrange cubic interpolation using GPU textures

Input: query coordinates $\mathbf{x}_q$, data texture tex of function $f(\mathbf{x})$

Output: interpolated value $f(\mathbf{x}_q)$

```

1:  $\mathbf{w} \leftarrow \text{lagrange\_weights}(\mathbf{x}_q)$ 
2:  $[\mathbf{g}, \mathbf{h}, \hat{\mathbf{x}}] \leftarrow \text{compute\_texture\_filter\_coord}(\mathbf{w}, \lfloor \mathbf{x}_q \rfloor)$ 
3:  $\text{trilinear}[1] \leftarrow \text{tex3D}(\text{tex}, \mathbf{h}, \hat{\mathbf{x}})$ 
4:  $\text{bilinear}[6] \leftarrow \text{tex3D}(\text{tex}, \mathbf{h}, \hat{\mathbf{x}})$ 
5:  $\text{linear}[12] \leftarrow \text{tex3D}(\text{tex}, \mathbf{h}, \hat{\mathbf{x}})$ 
6:  $\text{point}[8] \leftarrow \text{tex3D}(\text{tex}, \mathbf{h}, \hat{\mathbf{x}})$ 
7:  $\mathbf{a} \leftarrow \text{linear\_weighting\_X}(\text{trilinear}, \text{bilinear}, \text{linear}, \text{point}, \mathbf{w}, \mathbf{g})$ 
8:  $\mathbf{b} \leftarrow \text{linear\_weighting\_Y}(\mathbf{a}, \mathbf{w}, \mathbf{g})$ 
9:  $f(\mathbf{x}_q) \leftarrow \text{linear\_weighting\_Z}(\mathbf{b}, \mathbf{w}, \mathbf{g})$ 
```

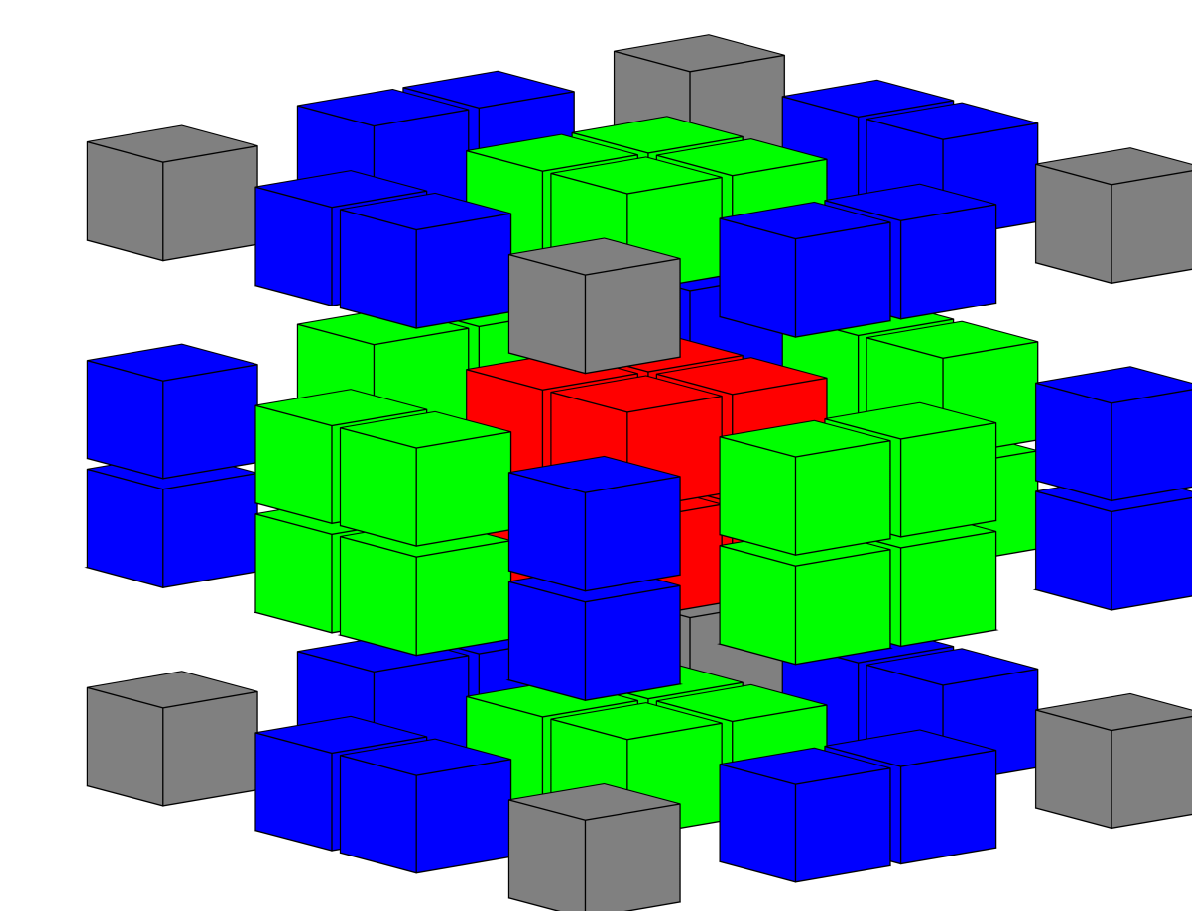


Figure 3: Illustration of Lagrange texture interpolation using 64 nearest neighbours. The **center core** consisting of 8 data points is used for a single trilinear texture lookup. The **6 faces** each consisting of 4 points are used for bilinear lookups. The **12 edges** each consisting of 2 points are used for linear lookups and the remaining 8 points are fetched using texture point lookup instead of accessing them from global memory. The resulting interpolated values are linearly weighted along x, y and z directions in that order.

Results

- We analyze the performance of our interpolation kernel on the Piz Daint system at the Swiss National Supercomputing Center. Each XC50 compute node of Piz Daint has a NVIDIA Tesla P100 GPU. We also perform additional runs on the Maverick system at the Texas Advanced Computing Center each node of which has a NVIDIA Tesla K40 GPU.
- We compare the speed and accuracy of our kernel with similar existing kernels [1, 2, 3, 4] for synthetic images of resolutions 64^3 and 256^3 . The results are shown in table 1.
- We also compare the performance of our algorithm with a CPU based implementation. The CPU benchmark is run on a single Haswell, Skylake and IvyBridge node separately with 20 MPI tasks each.

Variable	Achieved	Theoretical	Device Limit	Grid Size: [65536,1,1] (65536 blocks) Block Size: [256,1,1] (256 th
Occupancy Per SM				
Active Blocks		6	32	
Active Warps	42.42	48	64	
Active Threads	1536	2048		
Occupancy	66.3%	75%	100%	
Warps				
Threads/Block		256	1024	
Warps/Block		8	32	
Block Limit		8	32	
Registers				
Registers/Thread		39	65536	
Registers/Block		10240	65536	
Block Limit		6	32	

Figure 4: Theoretical and Observed occupancy per Streaming Multiprocessor for the most optimal thread block configuration obtained for the Lagrange texture interpolation kernel (for image size 256^3). Figures obtained from the NSight Visual Profiler.

Roofline Model Analysis

- We estimate the algorithmic Arithmetic Intensity (AI) of our kernel by computing the in-app Floating Point Operations (FLOPs) per byte of memory accessed perform a basic roofline model analysis with 'no cache' model.

- For a comparison, we also use the NVIDIA Visual Profiler to get the actual number of bytes moved from device memory to get a more accurate estimate of the Arithmetic Intensity. The analysis suggests that our kernel is memory bound. See figure 5

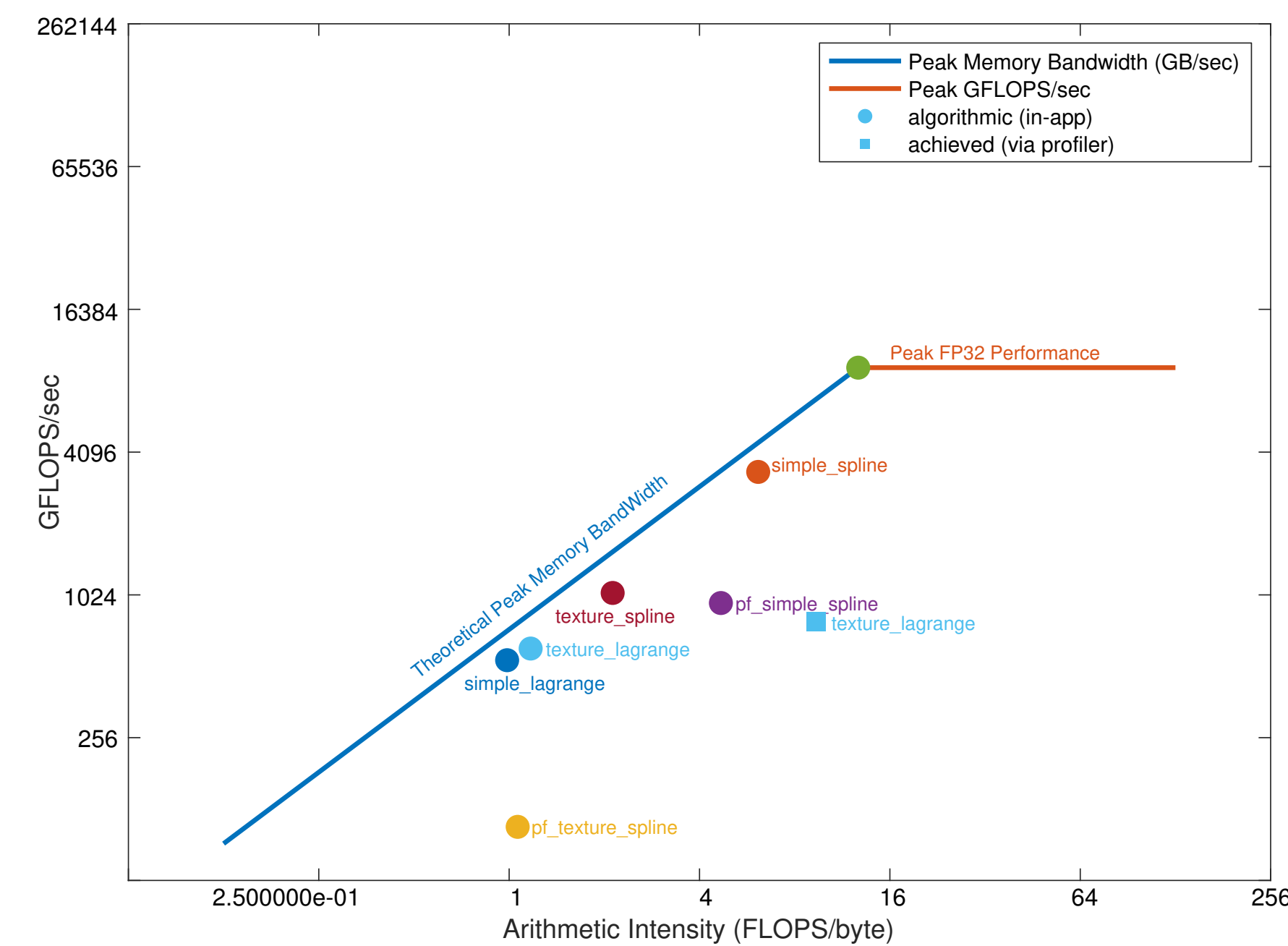


Figure 5: Roofline Analysis We compare different existing interpolation kernels with our implementation. A texture prefix implies hardware linear interpolation is used. A simple prefix implies texture memory is used as a direct lookup instead of utilizing hardware interpolation. A pt prefix implies prefiltering operation was done on image data prior to interpolation. The analysis indicates that all the kernels are primarily memory bound except for the prefilter kernel which is latency bound.

Fixed Departure Points vs Random Departure Points

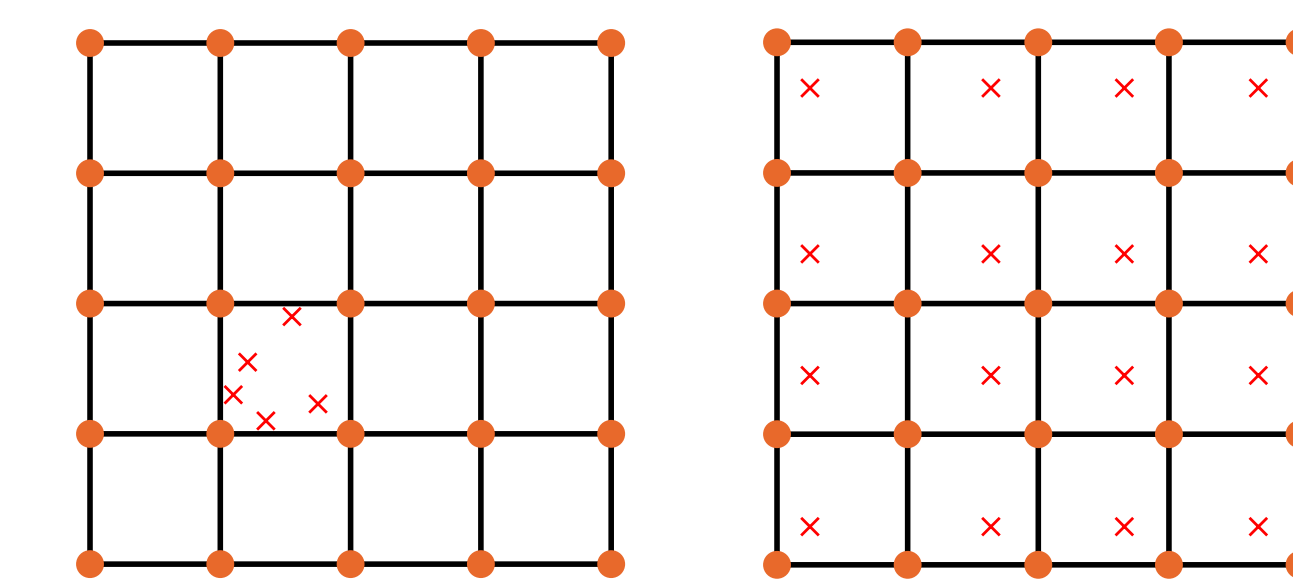


Figure 6: (left) fixed departure points (or interpolation coordinates) situated randomly within a single voxel, (right) random departure coordinates. Red stars show where interpolation is to be performed.

- To test the effectiveness of reusability of texture memory loads and to put a ceiling on the performance of the algorithm we conduct a simple experiment.
- Distribute the query coordinates in a single voxel to only have to load 64 neighbouring grid values once.
- For the left image in figure 6, we compare 2 kernel implementations. First, the Lagrange interpolation kernel where texture is used to access grid values and second, where the 64 grid values are loaded only once into shared memory.
- Shared memory implementation is $\sim 7\times$ faster than the texture lookup method, indicating very little reuse of texture loaded data. A shared memory implementation for a general case can prove to be highly effective.

Benchmarks on NVIDIA Tesla P100 and Tesla K40

Table 1: Comparison of computational performance of semi-lagrangian solver using different interpolation kernels in single precision on CPU and GPU. We report relative error and total execution timing (14 interpolations, excluding one-time device-host memory transfer) for the semi-lagrangian solver using a synthetic image m_s . The L_2 relative error is computed between m_s and the image obtained by solving eq 1 first using synthetic velocity $\mathbf{v}_s$ and then $-\mathbf{v}_s$.

resolution	64 ³	256 ³	64 ³	256 ³
kernel	rel. error	runtime(ms)	rel. error	runtime(ms)
intel haswell	2.16E−4	5.64	2.06E−4	344
intel skylake	2.16E−4	6	2.06E−4	321
intel ivyBridge	2.16E−4	5.18	2.06E−4	354
Tesla P100				
pf simple spline	2.93E−4	5.77	2.05E−4	421
pf texture spline	3.68E−4	4.27	2.08E−4	331
simple spline	2.17E−2	1.85	1.41E−3	115
simple lagrange	2.16E−4	1.79	2.06E−4	116
texture lagrange	3.88E−4	0.86	2.16E−4	56
texture spline	2.17E−2	0.36	1.41E−3	23
Tesla K40m				
pf simple spline	2.93E−4	5.77	2.05E−4	421
pf texture spline	3.68E−4	4.27	2.08E−4	331
simple spline	2.17E−2	1.85	1.41E−3	115
simple lagrange	2.16E−4	1.79	2.06E−4	116
texture lagrange	3.88E−4	0.86	2.16E−4	56
texture spline	2.17E−2	0.36	1.41E−3	23

Conclusions

We present a novel algorithm for Lagrange cubic interpolation for 3D images which utilizes texture memory. It is almost as accurate as a prefiltered cubic-BSpline interpolation and avoids the hassle of any prefiltering due to the nature of the Lagrange polynomials. We implement the Semi-Lagrangian time stepping scheme using this kernel and achieve good performance for single GPU case.

References

- P. THÉVENAZ, T. BLU AND M. UNSER, *Interpolation revisited [medical images application]*, IEEE Transactions on Medical Imaging, vol. 19-7 (2000), pp. 739–758.
- D. RUIJTERS AND P. THÉVENAZ, *GPU Prefilter for Accurate Cubic B-Spline Interpolation*, Journal of Computing, vol. 55-1, January 2012, pp. 15–20
- C. SIGG AND M. HADWIGER, *Fast Third-Order Texture Filtering* GPU Gems 2, February 2005, pp. 313–329
- D. RUIJTERS, BART. ROMENY AND P. SUETENS *Efficient GPU-based texture interpolation using uniform B-splines*, Journal of Graphic Tools, vol. 13-4, pp. 61–69, 2008, AK Peters Ltd.