Master’s Thesis
Performance Evaluation of the Intel Many Integrated
Core Architecture for 3D Image Reconstruction in
Computed Tomography

By
Johannes Hofmann

Supervision
Prof. Dr. rer. nat. Gerhard Wellein
Dr.-Ing. Jan Treibig

October 30, 2013
Abstract

The computational effort of 3D image reconstruction in Computed Tomography (CT) has required special purpose hardware for a long time. Systems such as custom-built FPGA-systems and GPUs are still widely-used today, in particular in interventional settings, where radiologists require a hard time constraint for reconstruction. However, recently it has been shown that today even commodity CPUs are capable of performing the reconstruction within the imposed time-constraint.

In this thesis, we examine the Intel Many Integrated Cores (MIC) architecture for its suitability to run the Feldkamp-Davis-Kress (FDK) algorithm—the most commonly used algorithm to perform the 3D image reconstruction in cone-beam computed tomography. In comparison to traditional CPUs the MIC accelerator card, which focuses on numerical applications, is expected to deliver higher performance using the same programming models such as C, C++, and Fortran.

A thorough analysis of the MIC architecture is performed to determine potential hardware bottlenecks and to distinguish its design from a current state of the art two-socket Intel Sandy Bridge EP CPU system.

We study the challenges of efficiently parallelizing the FDK kernel on the Intel MIC and find that careful OpenMP scheduling and thread placement is required due to lack of a shared last level cache. Efficient data sharing on the Intel MIC can only occur between hardware threads of a core via its local L1 and L2 cache segments.

Apart from parallelization, SIMD vectorization is critical for good performance on the Intel MIC, whose vector registers are twice the size of vector registers found in contemporary CPUs. To classify the difficulty of harnessing the full potential of vectorization on the MIC platform we explore various approaches to vectorize the kernel: Auto-vectorization using the Intel C Compiler and the Intel SPMD Compiler, as well as manual vectorization using C with intrinsics and manual assembly coding.

We used the fastest available CPU implementation from Treibig et al., developed for the RabbitCT benchmarking framework, as starting point for our optimizations. By making improvements to the original implementation, we speed up execution by 25% on the CPU. In line with the estimate of our performance model, measurements on the Intel MIC deliver a speedup of 1.5 in comparison to the reference CPU system. Our analysis reveals the major bottleneck for our application to be shortcomings in hardware: The majority of data required for the reconstruction is scattered in memory; gathering this data into vector registers for processing is still done sequentially on the Intel MIC. While computations in the kernel benefit from vectorization, the sequential loading limits the maximum achievable speedup in accordance with Amdahl's law.
# Contents

1. Introduction                                     1
   1.1. Computed Tomography ........................................ 1
   1.2. Intel's Xeon Phi ........................................... 3
   1.3. Related Work .................................................. 5
   1.4. Scope and Main Contribution ............................... 6
   1.5. Outline .......................................................... 7

2. Introduction to Computed Tomography ................. 9
   2.1. Tomograph Setup .............................................. 9
   2.2. RabbitCT Benchmarking Framework ........................... 15
   2.3. The Feldkamp-Davis-Kress Reconstruction Algorithm .... 17

3. Intel 64 and Intel Many Integrated Core Architectures 21
   3.1. Preliminaries .................................................. 21
       3.1.1. Environment ................................................ 21
       3.1.2. Likwid ..................................................... 22
       3.1.3. STREAM Benchmark ......................................... 22
       3.1.4. Plots ....................................................... 23
   3.2. Intel's Sandy Bridge Microarchitecture ............... 23
       3.2.1. Core Pipeline ............................................... 24
       3.2.2. Cache Organization, Core Interconnect, and Memory .... 27
       3.2.3. Advanced Vector Extensions ............................. 31
   3.3. Intel's Many Integrated Core Microarchitecture ..... 34
       3.3.1. Core Pipeline ............................................. 35
       3.3.2. Cache Organization, Core Interconnect, and Memory .... 37
       3.3.3. Initial Many Core Instructions ......................... 43
       3.3.4. Vector Gather Operation ................................ 44

4. Fastrabbit on x86 ............................................ 49
   4.1. Existing Optimizations ...................................... 49
       4.1.1. Avoiding Redundant Calculations ....................... 49
       4.1.2. Zero Padding ............................................... 51
       4.1.3. Clipping Mask ............................................. 52
       4.1.4. Vectorization ............................................... 53
Contents

4.1.5. OpenMP Parallelization .................................................. 59
4.2. Additional Optimizations ................................................. 59
4.2.1. Clipping Mask ............................................................... 59
4.2.2. Instruction Scheduling ..................................................... 61
4.2.3. Parameter Handling ....................................................... 61
4.2.4. Results ........................................................................ 63

5. Implementations for Intel MIC ........................................... 65
5.1. OpenMP Parallelization ..................................................... 65
5.2. Intel C Compiler ............................................................... 68
5.3. Intel SPMD Program Compiler .......................................... 71
5.4. C with Intrinsics ............................................................... 73
5.5. IMCI Assembly ................................................................. 78

6. Performance Model and Validation .................................. 87
6.1. Performance Model .......................................................... 87
6.1.1. Core Execution Time ....................................................... 87
6.1.2. Cache and Memory Subsystem Contribution .................... 89
6.2. Validation ....................................................................... 91

7. Results and Discussion ..................................................... 93

8. Conclusion and Outlook ................................................... 95

A. Register Mapping in the IMCI Line Update Kernel ............. 103
B. Measuring the Number of vgatherdps Instructions in a Gather Loop 105
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABI</td>
<td>Application Binary Interface</td>
</tr>
<tr>
<td>AES-NI</td>
<td>Advanced Encryption Standard – New Instructions</td>
</tr>
<tr>
<td>AVX</td>
<td>Advanced Vector Extensions</td>
</tr>
<tr>
<td>BPU</td>
<td>Branch Prediction Unit</td>
</tr>
<tr>
<td>CL</td>
<td>Cache Line</td>
</tr>
<tr>
<td>CRI</td>
<td>Core-Ring Interconnect</td>
</tr>
<tr>
<td>CT</td>
<td>Computed Tomography</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DP</td>
<td>Double Precision</td>
</tr>
<tr>
<td>FBP</td>
<td>Filtered Back Projection</td>
</tr>
<tr>
<td>FDK</td>
<td>Feldkamp-Davis-Kress</td>
</tr>
<tr>
<td>GUP/s</td>
<td>Giga Voxel Updates per Second</td>
</tr>
<tr>
<td>IACA</td>
<td>Intel Architecture Code Analyzer</td>
</tr>
<tr>
<td>IF</td>
<td>Instruction Fetcher</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
</tr>
<tr>
<td>IMCI</td>
<td>Initial Many Core Instructions</td>
</tr>
<tr>
<td>ISPC</td>
<td>Intel SPMD Program Compiler</td>
</tr>
<tr>
<td>L1</td>
<td>Level One</td>
</tr>
<tr>
<td>L2</td>
<td>Level Two</td>
</tr>
<tr>
<td>LCPs</td>
<td>Length Changing Prefixes</td>
</tr>
<tr>
<td>LLC</td>
<td>Last Level Cache</td>
</tr>
<tr>
<td>MIC</td>
<td>Many Integrated Cores</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean Squared Error</td>
</tr>
<tr>
<td>MSR</td>
<td>Model Specific Registers</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Access</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Open Computing Language</td>
</tr>
<tr>
<td>PF</td>
<td>Picker Function</td>
</tr>
<tr>
<td>PSNR</td>
<td>Peak Signal-to-Noise Ratio</td>
</tr>
<tr>
<td>QPI</td>
<td>QuickPath Interconnect</td>
</tr>
<tr>
<td>SHA</td>
<td>Secure Hashing Algorithm</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data</td>
</tr>
<tr>
<td>SM</td>
<td>Streaming Multiprocessor</td>
</tr>
<tr>
<td>SP</td>
<td>Single Precision</td>
</tr>
<tr>
<td>SSE</td>
<td>Streaming SIMD Extensions</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>SVML</td>
<td>Short Vector Math Library</td>
</tr>
<tr>
<td>TD</td>
<td>Tag Directory</td>
</tr>
<tr>
<td>TFlop/s</td>
<td>Tera Floating-Point Operations per Second</td>
</tr>
<tr>
<td>TTS</td>
<td>Time-to-Solution</td>
</tr>
<tr>
<td>VPU</td>
<td>Vector Processing Unit</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1. Computed Tomography

Up until 1895, the only way to examine the inner structure of objects was to open and disassemble them. This invasive process often caused damage or even destroyed the object of interest. Wilhelm Conrad Röntgen’s discovery of x-rays [1] fundamentally changed this premise, making it one of most important advances in both medicine and non-destructive material testing.

In the early years after its inception, the technology was used to create single x-ray images of objects. In order to get an idea of an object’s inner three-dimensional structure, multiple images had to be acquired form different perspectives. It was then left to the imagination of the examiner to envision the inner design of the considered object.

This initial drawback was later overcome: Although the theoretical means to construct a 3D volume from a set to 2D images has been available since 1917 with the Radon transform [2], a way to compute the solution in feasible time only became available much later with the proliferation of computers. With increasing mechanization of acquisition techniques and the continuing rise in computing performance it was possible to completely automate the reconstruction, i.e. to transform the information from the set of acquired 2D images into a 3D volume.

Today, this automated procedure, known as [CT], is used in a wide variety of applications. In industrial CT, the method is used to perform non-destructive material testing: Imagine some expensive cast iron part that is scheduled for material fatigue testing. If the part does not suffer from any problems, we would like to continue using it. Since opening the part for the examination is not always an option—the process of opening could be destructive in such a way that the part can no longer be used afterwards—computed tomography can provide a solution to the problem.

Apart from industrial usage, computed tomography is also widely used in medical applications, where it can be either used as a diagnostic or interventional tool.

In the diagnostic case, the device is used to examine patients with almost no time pressure. To perform his diagnosis, the treating medical doctor—a general practitioner or medical specialist—is referring the patient to a radiologist, who actually performs the CT scan.
1. Introduction

The CT scan is then sent back to the transferring doctor, who can then make his diagnosis. Although the results of the CT scan should reach the transferring doctor in a timely manner (within days), in the diagnostic form there is no hard medical time-constraint necessitating a fast reconstruction.

This situation is different when computed tomography is used in interventional settings, e.g. for a patient with an aneurysm. Today, instead of performing open-brain surgery to clamp such an aneurysm, the use of interventional CT presents the possibility to offer a much less invasive treatment. Using CT, a catheter can be inserted into the patient’s body in the groin area and guided through the torso to the bottom of the brain; the guidance of the catheter tip—which is made of metal, providing good contrast with tissue—is performed using the visualization of a 3D volume, reconstructed from the data taken with an interventional C-arm CT device (see Figure 1.1). Since the position of the catheter tip has to be verified and corrected quite frequently along the way from the groin to the brain, it is imperative the reconstruction can be performed in a matter of seconds.

Figure 1.1.: Axium Artis by Siemens, a wall-mounted interventional C-arm CT device. Image courtesy of Siemens AG.

Several years ago, the task to perform this fast reconstruction was reserved for a limited range of special-purpose hardware, such as high-bandwidth FPGAs. The shortcomings of

---

An aneurysm is a blood-filled bulge in the wall of a blood vessel, which commonly occurs in arteries at the base of the brain.
this solution were that the FPGAs were not only costly in terms of actual hardware, but also because developing and maintaining software or VHDL designs for this kind of special-purpose hardware is a time- and therefore cost-intensive enterprise as well.

With the advent of powerful graphics processing units (GPUs) and a means to program these devices in a simple manner (e.g. CUDA, OpenCL), this trend has changed—Siemens, for example, released their first GPU-based interventional C-arm CT in 2010. Today, manufacturers of C-arm CT devices rely on these GPUs to perform the fast reconstruction, because they are both cheaper in acquisition and easier to program than the special-purpose hardware used before. An additional benefit of GPUs is that they are much faster than the hardware used before. Nvidia’s latest “Kepler” architecture can perform the back projection part of the reconstruction—the most compute- and time-intensive part of reconstruction—in less than four seconds, more than about ten times faster than the previous FPGA hardware.

With the release of Intel’s Many Integrated Cores (MIC) architecture in late 2012, there exists a new competitor in the field of fast CT reconstruction. Intel’s hardware has approximately the same key performance figures as the fastest GPU from Nvidia’s latest Kepler architecture. However, apart from being on par performance-wise, Intel’s new architecture offers the benefit of being programmable by standard languages such as C, C++, and Fortran; to achieve the best performance on Nvidia’s GPUs, on the other hand, developers are restricted mainly to the CUDA programming model, which is vendor-specific. These reasons make the Xeon Phi—the official brand name for Intel’s MIC architecture—an interesting candidate, which is worth examining for the task of fast CT reconstruction.

1.2. Intel’s Xeon Phi

Just like GPUs, the Intel Xeon Phi is a PCIe-based accelerator card (see Figure), which, as Intel states, is meant to be used as a coprocessor to speed up highly parallel workloads. Unlike a GPU, it is programmable by an array of standardized languages (C, C++, and Fortran) and is compatible with the x86 instruction set architecture. While reusing existing binaries is not an option due to changes in the application binary interface, legacy codes can simply be recompiled and run on the Xeon Phi without any modification.

In a way, Intel has been evolving the final design since 2006; this was the time that Intel began with the development of a GPU chip, codenamed Larrabee, separate from its line of integrated graphics accelerators. However, a GPU featuring the chip never saw the light of day. Instead, in 2010 Intel decided that the results of the groundwork for Larrabee should find their way into the MIC architecture, which is the design used in the Xeon Phi coprocessor.

The card was developed to be—and turned out—a direct competitor for Nvidia’s Telsa

---

2While the official release of the hardware was announced at the SC12 in November 2012 and several installations such as Texas Advanced Computing Center’s Stampede were already equipped with the hardware at that time, it was well into 2013 until ordinary customers would receive the hardware.

3Although Nvidia’s GPUs are also programmable by the standardized OpenCL language, Nvidia sees to it that its own CUDA standard gets a preferential treatment (e.g. performance is slightly better, new features are always added to CUDA first, etc.).
1. Introduction

A direct comparison of some key performance figures has been compiled in Table 1.1. Values for Single Instruction, Multiple Data (SIMD) register width and peak Tera Floating-Point Operations per Second (TFlop/s) are given in both Single Precision (SP) and Double Precision (DP).

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Cores/Clock</th>
<th>SIMD</th>
<th>Peak TFlop/s</th>
<th>Peak BW</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon Phi 5110P</td>
<td>60/1.05GHz</td>
<td>16SP/8DP</td>
<td>2.02SP/1.00DP</td>
<td>320GiB/s</td>
<td>$2,600</td>
</tr>
<tr>
<td>Xeon Phi SE10P</td>
<td>61/1.10GHz</td>
<td>16SP/8DP</td>
<td>2.15SP/1.07DP</td>
<td>352GiB/s</td>
<td>—</td>
</tr>
<tr>
<td>Tesla K20</td>
<td>13/706MHz</td>
<td>192SP/64DP</td>
<td>3.52SP/1.17DP</td>
<td>208GiB/s</td>
<td>$3,000</td>
</tr>
<tr>
<td>Tesla K20x</td>
<td>14/732MHz</td>
<td>192SP/64DP</td>
<td>3.94SP/1.31DP</td>
<td>250GiB/s</td>
<td>$5,000</td>
</tr>
</tbody>
</table>

1 At the time of this writing (June 14, 2013).

Table 1.1: Key performance data of the Xeon Phi and Tesla K20 accelerators.

As we can see in this Table, both accelerators currently come in two flavors. The only officially released Xeon Phi hardware is the 5110P; the (special edition) SE10P card was reserved for some installations in supercomputers and as pre-release evaluation hardware. The special edition card is clocked slightly higher and has an additional core active. Both Tesla cards are based on the GK110 chipset. The ordinary K20 uses the poorer-quality chips, while the K20x is built using the flawless chips of a production batch. The K20x therefore manages to have one more core and an additional memory controller active.

Each of the Xeon Phi's cores features a 512 bit Vector Processing Unit (VPU), which is ca-
1.3. Related Work

Due to its medical relevance, reconstruction in computed tomography is a very well-examined problem. As vendors for CT devices are constantly on the lookout for ways to speed up the reconstruction time, it is hard to find a computer architecture that hasn’t been evaluated over time.

First implementations capable of meeting the time constraints required during medical procedures were using expensive special-purpose hardware like the ImageProX accelerator platform, which comes with up to eight dedicated FPGA chips.\(^5\) The Cell Broadband Engine, which at the time of its release provided unrivaled memory bandwidth, was also subject to experimentation.\(^6\) The Cell is a truly heterogeneous platform, featuring one PowerPC CPU acting as supervisor (running the operating system) supplemented by eight vector units, which are connected together via a high-bandwidth bi-

---

\(^5\)In accordance with the findings of Volkof and Demmel,\(^7\) we treat GPUs as multi-core vector units. In this view, the Streaming Multiprocessor units of a GPU correspond to cores of a CPU; the CUDA cores of each SM correspond of the SIMD-lanes of a CPU core’s vector registers.

\(^6\)Note that in the GDDR5 standard, the memory channel width is only 32 bits (as opposed to 64 bits in DDR3, currently used on CPUs).
1. Introduction

directional ring. A combination of its heterogeneity together with some other shortcomings (communication over the ring was only possible using explicit DMA accesses, lengthy to implement; program size was limited, because on the vector units program and data shared a small 256 KiB memory; free compilers were missing basic optimizations for the Cell platform, making tedious manual optimization necessary even for the simplest programs) and the lack of adequate development tools for the platform lead to the community quickly abandoning the Cell processor as soon as other hardware with similar performance capabilities such as GPUs arrived.

It is noteworthy that CT reconstruction was among the first non-graphics applications that were run graphics processors [10]. Although programming for graphics hardware was quite difficult and subject to a lot of limitations in the early days, today GPUs have evolved into truly programmable general-purpose processors while languages such as CUDA and OpenCL simplify the development for these modern GPUs. This led to a sheer endless number of implementations for GPUs being available today [11]. However, the use of varying data sets and reconstruction parameters limited the comparability of all these implementations. In an attempt to remedy this problem, the RabbitCT framework [12] provides a standardized, freely available CT scan data set and a uniform benchmarking interface that evaluates both reconstruction performance and accuracy.

Some of the current entries the RabbitCT ranking worth mentioning include Thumper by Zinsser and Keck[13], a Kepler-based implementation which currently dominates all other implementations and fastrabbit by Treibig et al. [14], a highly optimized CPU-based implementation, for which the authors found that vectorization is critical for good performance on modern CPU architectures.

However, as of this writing, no one has had the opportunity to implement the back projection on a Xeon Phi yet. We have thus used this occasion and implement the FDK algorithm for this device, examining the hardware in detail along the way.

1.4. Scope and Main Contribution

This thesis presents a detailed examination of the Intel MIC architecture and its application to the task of CT-reconstruction in interventional settings.

In the first part we present both the preliminaries to understand how CT reconstruction works and perform an in-depth review the CPU reference system as well as the MIC architecture. We perform a number of micro-benchmarks to evaluate the cache and memory subsystems of both architectures, as well as the new vector gather operation of the Intel MIC. We also devised a portable way to reliably measure instruction latencies in order to work around some blanks in the documentation from Intel.

Next, the various optimizations used in the original fastrabbit implementation [14]—an existing RabbitCT implementation for CPUs, which served as a starting point for the Intel MIC implementations—are presented. In this part, the main contributions are the improvement of the existing clipping mask optimization, as well as two additional optimizations involving parameter handling and register scheduling to speed up the reconstruction.

What follows is an investigation of efficient OpenMP multi-core parallelization on the Intel
1.5. Outline

MIC considering sensible OpenMP scheduling and thread placement to maximize L2 data sharing among a core’s hardware threads. A range of vectorization approaches for the Intel MIC is examined next. These include auto-vectorization using compilers as well as manual vectorization through intrinsics and writing assembly code. To validate the performance of these implementations we develop a performance model for the kernel used during the reconstruction on the Intel MIC.

1.5. Outline

The structure of this thesis is as follows. Chapter 2 introduces several of the ideas behind computed tomography (Section 2.1) and lays the groundwork to understand the Feldkamp-Davis-Kress reconstruction method [15] (Section 2.3), which is the prevalent technique to perform reconstruction in interventional settings; the need for a standardized benchmarking framework is motivated and the RabbitCT framework is described in Section 2.2.

Chapter 3 introduces Intel’s Sandy Bridge microarchitecture as the most recent representative x86 server CPU in Section 3.2. We introduce the general pipeline design, cache organization and core interconnect as well as standard vectorization capabilities (3.2.1–3.2.3). After having established the Sandy Bridge architecture as reference, we focus on examining the MIC architecture in detail, working out both major and minor differences between a standard CPU and the Intel MIC in Sections 3.3.1–3.3.4.

Chapter 4 introduces the implementation that served as a starting point for the Intel MIC. We describe all of the optimizations that were employed in the original version in Section 4.1 and continue with additional optimizations that were devised for the CPU version as part of this thesis in Section 4.2.

Chapter 5 begins with an analysis for effective OpenMP parallelization on the Intel MIC in Section 5.1. Next, we present the implementations that were devised for the accelerator. These include an implementation written in C, vectorized once by the Intel C compiler using compiler flags and once using the Intel C compiler together with new capabilities of the latest OpenMP 4 standard [16] (Section 5.2); an implementation for the Intel SPMD Program Compiler (ISPC) (Section 5.3); a version written in C using intrinsics (Section 5.4); and finally, an implementation written in assembly (Section 5.5).

Chapter 6 first establishes a detailed performance model for our application in Section 6.1; this model is then used to perform a validation of the results obtained with the assembly implementation in Section 6.2.

Chapter 7 contains a summary for all the implementations described in Chapter 5 and a discussion of the obtained results; this is followed by the conclusion in Chapter 8.
Chapter 2

Introduction to Computed Tomography

2.1. Tomograph Setup

When Godfrey Hounsfield created the first computer tomograph on behalf of EMI\textsuperscript{1} in 1971, he effectively set the \textit{modus operandi} for all subsequent CT devices: Source and flat-panel detector move along a defined trajectory—mostly a circle—around the volume that holds the object to be investigated and x-ray images are taken at regular angular increments along the way. The setup of such a computer tomograph is shown in Figure 2.1.

For medical applications, the reconstruction is almost exclusively performed using the Filtered Back Projection (FBP). Apart from the Radon transform, which accounts for the “back projection” part of the name, FBP also includes preprocessing of the acquired x-ray images (also known as projection images), which involves filtering and possibly cosine weighting. Despite being an integral part of the FBP, the algorithms involved in the preprocessing step are usually not part of optimization attempts, as they are less computationally complex—$O(n^2)$ compared to $O(n^3)$ of the Radon transform. Consequently, the rest of this Section will focus on the back projection part of the FBP.

In general, the back projection works by transforming the information recorded in the individual projection images into a 3D volume, which is made up of individual voxels (volume elements). To get the intensity value for a particular voxel and one of the recorded projection images we follow the x-ray that is projected from the x-ray source through the isocenter of the voxel to the detector; the intensity value at the resulting detector coordinates is then read from the recorded projection image and added to the voxel. This process is done for each voxel of the volume and all recorded projection images.

In order to perform this forward-projection from the x-ray source through the volume’s voxels onto the detector, we have to convert between the different geometries of the tomograph. An overview of all relevant geometries is shown in Figure 2.2.

The camera coordinate system describes the position and orientation of the x-ray source, which changes its position along the acquisition trajectory during the recording of the projection images.

\textsuperscript{1}Some people claim that the enormous success of the Beatles, signed under the EMI label, and the associated financial benefit for the label was what made this invention possible.
2. Introduction to Computed Tomography

Figure 2.1.: Ubiquitous setup for computer tomographs with circular acquisition trajectory: The x-ray source on one and the x-ray detector on the other side rotate around the volume, which holds the object of interest. The reconstruction is based on the projection images, which are recorded on the detector at fixed intervals.

Figure 2.2.: Important geometries of the computer tomograph setup: Camera Coordinate System (CCS), Voxel Coordinate System (VCS), World Coordinate System (WCS), Detector Coordinate System (DCS), and Image Coordinate System (ICS).
The voxel coordinate system is not of direct relevance for the computer tomograph setup; it could be called a virtual or logical coordinate system and is used during reconstruction to facilitate addressing voxels in computer memory.

The world coordinate system is located in the isocenter of the volume to be reconstructed and serves as reference coordinate system when projecting an x-ray from the source through the volume onto the detector.

The detector coordinate system is located at the center of the detector and serves as reference when converting to the image coordinate system—the logical coordinate system used to address the pixels’ intensity values in computer memory.

After introducing the different coordinate systems, we can now focus on the actual conversions between them. We start with the camera coordinate system.

\[ \begin{bmatrix} x_w \\ y_w \\ z_w \\ \end{bmatrix} = R \begin{bmatrix} x_{w'} \\ y_{w'} \\ z_{w'} \\ \end{bmatrix} + \vec{t} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{w'} \\ y_{w'} \\ z_{w'} \\ \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \] (2.1)

Using homogeneous coordinates, which extend the Cartesian coordinate vector by an additional element, we can transform the affine mapping into a linear mapping:

\[ D \begin{bmatrix} x_{w'} \\ y_{w'} \\ z_{w'} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & t_x \\ \sin \theta & \cos \theta & 0 & t_y \\ 0 & 0 & 1 & t_z \end{bmatrix} \begin{bmatrix} x_{w'} \\ y_{w'} \\ z_{w'} \\ 1 \end{bmatrix} = \begin{bmatrix} hx_{w} \\ hy_{w} \\ hz_{w} \\ h \end{bmatrix} \] (2.2)
After this operation, we have to dehomogenize the resulting homogeneous coordinate vector to obtain an unique set of Cartesian coordinates. We do that by first normalizing the vector, i.e. dividing by the extended homogeneous coordinate $h$, and then removing the extended coordinate:

$$
\begin{bmatrix}
h x_w \\
h y_w \\
h z_w \\
h
\end{bmatrix}
\rightarrow
\begin{bmatrix}
x_w \\
y_w \\
z_w \\
1
\end{bmatrix}
\rightarrow
\begin{bmatrix}
x_w \\
y_w \\
z_w
\end{bmatrix}
$$

(2.3)

After establishing how to correct for extrinsic camera parameters, we can now address the forward-projection, i.e. the process of how to determine a voxel’s intensity value in the recorded projection image. We start with voxel coordinates, which are converted to world coordinates; the world coordinates are then projected onto the detector to receive detector coordinates; in the final step we convert the detector coordinates into image coordinates.

A sample $2 \times 2 \times 2$ volume, which we will use to describe the conversion step from voxel to world coordinates, is shown in Figure 2.4.

![Figure 2.4: 2 × 2 × 2 volume with the world coordinate system shown in red at the center of the volume; the voxel coordinate system is shown in blue on the bottom left; the translation of the WCS using the offset $O$ is shown in green.](image)

The first step in converting from voxel coordinates to world coordinates is to determine the offset $O$ that is used to express the position of the isocenter of voxel $(0, 0, 0)$ (shown in green in Figure 2.4) in world coordinates. This offset can be calculated using two steps:

- First, we map the WCS onto the VCS: Since each edge of the volume has length $l$, this is done by subtracting $l/2$ from the center of the world coordinate system.
• In a second step, we need to add half the voxel length MM to each component to arrive at the isocenter of the voxel, which is where the projected x-ray is supposed to pass through the voxel. With the volume edge length \( l \) and \( L \) voxels along each edge, the voxel length is given by \( MM = l/L \).

Combining these steps yields the offset \( O = -l/2 + MM/2 \). Using this initial offset, we can convert each voxel’s coordinates \((x_v, y_v, z_v)\) into world coordinates \((x_w, y_w, z_w)\) by scaling the voxel’s logical coordinates (VCS) with a voxel’s length \( R \) in the world coordinate system:

\[
\begin{bmatrix}
x_w' \\
y_w' \\
z_w'
\end{bmatrix} = \begin{bmatrix}
O + x_vMM \\
O + y_vMM \\
O + z_vMM
\end{bmatrix}
\]

(2.4)

Note that since the logical addressing of voxels (VCS) is unaware of the camera position, this conversion leaves us with the initial world coordinates \((x_w', y_w', z_w')\) described earlier. We correct these coordinates using Equations 2.2 and 2.3.

To convert from the camera-adjusted world coordinates \((x_w', y_w', z_w')\) to the detector coordinates \((x_d, y_d)\), we make use of the intercept theorem. We illustrate the conversion using the \( y_w \) coordinate (see Figure 2.5):

\[
\frac{z_w}{d + y_w} = \frac{y_d}{D} \Rightarrow y_d = \frac{D \cdot z_w}{d + y_w},
\]

(2.5)

with \( d \) the (fixed) distance from the origin of the camera coordinate system (position of x-ray source) to the origin of the world coordinate system (isocenter of reconstruction volume) and \( D \) the (fixed) distance from the origin of the camera coordinate system to the detector coordinate system (center of the flat panel detector). As the center of the WCS lies at the isocenter of the volume to be reconstructed, we can simply add the value of the signed \( y_w \) coordinate to \( d \).

We can use the exact same reasoning to calculate the \( x_d \) coordinate; then, again using homogeneous coordinates, we can construct the matrix \( P \) which allows for a linear mapping to make the implied conversion:

\[
P \begin{bmatrix}
x_w' \\
y_w' \\
z_w'
\end{bmatrix} = \begin{bmatrix}
D/d & 0 & 0 & 0 \\
0 & 0 & D/d & 0 \\
0 & 1/d & 0 & 1
\end{bmatrix} \begin{bmatrix}
x_w \\
y_w \\
z_w \\
1
\end{bmatrix} = \begin{bmatrix}
h x_d \\
h y_d \\
h \\
1
\end{bmatrix} \Rightarrow \begin{bmatrix}
x_d \\
y_d \\
1
\end{bmatrix}
\]

(2.6)

The last step left is to convert from detector coordinates \((x_d, y_d)\) to image coordinates \((x_i, y_i)\), which can then be used to access the intensity value in memory.

We construct a matrix \( K \), which will in turn be made up by several matrices, implementing the operations needed to perform the mapping (see Figure 2.8):

• A rotation matrix \( R \) corrects some of the imperfections in device setup, such as a possible skew angle between axis;

• a matrix \( S \) that—if necessary—corrects pixel extents with appropriate scaling factors \( p_x, p_y \); and
2. Introduction to Computed Tomography

Figure 2.5.: Side view of the computer tomograph setup illustrating perspective projection correction based on the intercept theorem.

Figure 2.6.: A skewed flat-panel detector with non-fixed ratio pixels.
2.2. RabbitCT Benchmarking Framework

- a final translation \( \vec{t} \) that moves the center of the detector coordinate system onto the image coordinate system.

We first combine matrices \( R \) and \( S \) into a single matrix \( Q \):

\[
Q = R \cdot S = \begin{bmatrix}
\cos \Theta & -\sin \Theta & 1/p_x & 0
\sin \Theta & \cos \Theta & 0 & 1/p_y
\end{bmatrix} = \begin{bmatrix}
\cos \Theta/p_x & -\sin \Theta/p_y
\sin \Theta/p_x & \cos \Theta/p_y
\end{bmatrix} \quad (2.7)
\]

Taking the additional translation, which moves the center of the detector coordinate system onto the image coordinate system, into account, we can specify the whole conversion from detector coordinates to image coordinates as:

\[
Q \begin{bmatrix} x_d \\ y_d \end{bmatrix} + \vec{t} = \begin{bmatrix}
\cos \Theta/p_x & -\sin \Theta/p_y \\
\sin \Theta/p_x & \cos \Theta/p_y
\end{bmatrix} \begin{bmatrix} x_d \\ y_d \end{bmatrix} + \begin{bmatrix} -O_x \\ -O_y \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad (2.8)
\]

We can again use homogeneous coordinates to combine the affine translation into a linear transformation using a matrix \( K \):

\[
K \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = \begin{bmatrix}
\cos \Theta/p_x & -\sin \Theta/p_y & -O_x \\
\sin \Theta/p_x & \cos \Theta/p_y & -O_y \\
0 & 0 & 1
\end{bmatrix} \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = \begin{bmatrix} hx_i \\ hy_i \\ h \end{bmatrix} \cong \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \quad (2.9)
\]

The total perspective transformation \( A \) can now be defined as the successive application of matrix \( D \), encoding the extrinsic camera parameters, matrix \( P \), encoding the projection model, and matrix \( K \), encoding the intrinsic camera parameters:

\[
D \cdot P \cdot K = A \in \mathbb{R}^{3 \times 4} \quad (2.10)
\]

Using this projection matrix, we can easily convert from world to image coordinates using homogeneous coordinates:

\[
A \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix}
a_0 & a_3 & a_6 & a_9 \\
a_1 & a_4 & a_7 & a_{10} \\
a_2 & a_5 & a_8 & a_{11} \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} hx_i \\ hy_i \\ h \end{bmatrix} \cong \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \quad (2.11)
\]

Note that the homogeneous image coordinates are also referred to as \( u, v, \) and \( w \); this naming is used throughout all source code listings.

2.2. RabbitCT Benchmarking Framework

Comparing different optimized back projection implementations found in the literature with respect to their performance can be difficult, because of variations in data acquisition and preprocessing, as well as different geometry conversions and the use of proprietary data sets.
2. Introduction to Computed Tomography

The RabbitCT framework [12] was designed as an open platform that tries to remedy the previously mentioned problems. It features a benchmarking interface, a prototype back projection implementation, and a filtered, high resolution CT dataset of a rabbit (see Figure 2.7); also included is a reference volume that is used to derive various image quality measures. All the findings presented in the remainder of this work are based on benchmarks obtained with the RabbitCT framework.

Figure 2.7.: Left: One of the preprocessed, i.e. parker-weighted and ramp-filtered, projection images from the RabbitCT dataset. Right: A ray-traced volumetric representation of the reconstructed 3D volume produced from the RabbitCT projections.

The preprocessed dataset consists of $N = 496$ projection images that were acquired using a commercial C-arm CT system. Each projection is $1248 \times 960$ pixels wide and stores the already filtered and weighted intensity values as single-precision floating-point numbers. In addition, each projection comes with a projection matrix $A \in \mathbb{R}^{3 \times 4}$ (see Equation 2.10), which can be used to perform the necessary perspective corrections from world coordinates to image coordinates.

The framework takes care of all required steps to set up the benchmark, so the programmer can focus entirely on the actual back projection implementation, which is provided as a module (shared library) to the framework. This module has to implement several functions:

- The $\text{RCTLoadAlgorithm}$ and $\text{RCTPrepareAlgorithm}$ functions are supposed to deal with all kinds of data initialization that has to happen before the back projection can begin. For example, this can involve allocating and initializing memory for the volume data.

- The $\text{RCTAlgorithmBackprojection}$ function contains the actual back projection implementation and is called once for each projection image. Only the runtime of this function is measured and included in the performance results.

- The $\text{RCTFinishAlgorithm}$ and $\text{RCTUnloadAlgorithm}$ functions are for housekeeping. The former is called after the last projection image has been processed, i.e. right before
the image quality measures are taken; the latter is called directly afterwards, tasks performed may include freeing the memory allocated before and/or during the reconstruction.

To objectively compare different back projection implementations, the framework makes use of four different evaluation metrics:

- The average runtime $t_{\text{avg}}$ of $\text{RCTAlgorithmBackprojection}$, which represents the average time for one call of the $\text{RCTAlgorithmBackprojection}$ function, i.e. the average time to process all $L^3$ voxels for a single projection image.

- The Mean Squared Error (MSE) $q_{\text{mse}}$ of the reconstructed volume $V$ in comparison to the reference reconstruction $V_{\text{ref}}$ that comes with the data set, which is defined as:

$$q_{\text{mse}} = \frac{1}{L^3} \sum_{x,y,z} [V(x, y, z) - V_{\text{ref}}(x, y, z)]^2$$  \hspace{1cm} (2.12)

- The Peak Signal-to-Noise Ratio (PSNR) $q_{\text{psnr}}$, which is based on the MSE $q_{\text{mse}}$:

$$q_{\text{psnr}} = 10 \log_{10} \left( \frac{(2^{12} - 1)^2}{q_{\text{mse}}} \right)$$  \hspace{1cm} (2.13)

- Additionally, a histogram containing the absolute errors in comparison to the reference volume is computed.

After executing a reconstruction using the RabbitCT framework, a result file containing all the metrics mentioned above is created; these results can then be published on the project homepage [17].

The next Section will introduce the theory and a slightly modified implementation of the reference back projector that comes with the framework.

2.3. The Feldkamp-Davis-Kress Reconstruction Algorithm

Today, the FDK [15] is the most successful FBP implementation and is used in almost all commercially available computer tomographs. The standard version of the algorithm deals with a circular acquisition trajectory, although the FDK method can be modified to deal with a spiral trajectory as well.

From a high-level viewpoint, the FDK method works by back projecting the distance-weighted, bilinearly interpolated intensity values for each voxel $(x_v, y_v, z_v)$ in the volume $V$ for all acquired projection images $I_n$:

$$V(x_v, y_v, z_v) = \sum_{n=1}^{N} \frac{1}{w^2} b_n(x_i, y_i),$$  \hspace{1cm} (2.14)

\footnote{The seemingly arbitrary constant of $2^{12} - 1$ in the Equation stems from the fact that the intensity values recorded by the flat-panel detector are normalized to lie in the range of 0 to $(2^{12} - 1)$.}
with $w$ the weighting factor to perform attenuation according to the inverse-squares law and $b_n$ the bilinear interpolation in projection image $I_n$, defined as:

$$b_n = \alpha'\beta' I_n(x'_i, y'_i) + \alpha\beta I_n(x'_i + 1, y'_i) + \alpha'\beta I_n(x'_i, y'_i + 1) + \alpha\beta I_n(x'_i + 1, y'_i + 1), \quad (2.15)$$

with image addressing offsets $x'_i = \lfloor x_i \rfloor$; $y'_i = \lfloor y_i \rfloor$ and interpolation weights $\alpha = x_i - x'_i$, $\alpha' = 1 - \alpha$, $\beta = y_i - y'_i$, $\beta' = 1 - \beta$. The conversion from voxel to image coordinates is performed according to Equations 2.4 and 2.11.

The straight-forward transformation of this mathematical representation into source code can be seen in Listing 2.1. This reference code is called once for every projection image. Note that the source code shown here does not represent the complete RCTAlgorithmBackprojection function, but only contains the reconstruction kernel; parameter handling is not shown and is to be done dealt with before the execution of the kernel. Therefore, we assume that $A$ holds the current projection matrix $A_n$ and $I$ holds the current projection image $I_n$.

The three outer for loops (lines 1–3) are used to iterate over all voxels in the volume. The loop variables $x$, $y$, and $z$ correspond to the logical voxel coordinates ($x_v$, $y_v$, $z_v$).

The kernel begins with the conversion of these logical voxel coordinates to world coordinates in lines 5–7 according to Equation 2.4. The variables $O$ and $MM$—containing the offset $O$ and voxel size $MM$—that are required to perform this conversion are precalculated by the RabbitCT framework and made available to the back projection implementation in a struct pointer that is passed to the RCTAlgorithmBackprojection function as a parameter.

After this, making use of the projection matrix $A_n$, the world coordinates are converted to image coordinates in lines 9–11; subsequently, the coordinates are dehomogenized, i.e. normalized to two unique Cartesian coordinates, in lines 13–14 (cf. Equation 2.11).

In the next step, we need to perform the bilinear interpolation according to Equation 2.15. In order to do so, we have to convert the image coordinates from floating-point format to integer format (lines 16–17), because we need integer values to address the intensity values in the projection image buffer $I$. The interpolation weights $\text{scalex}$ and $\text{scaley}$ are calculated in lines 19–20 and correspond to the weights $\alpha$ and $\beta$ of the mathematical description.

The four values needed for the bilinear interpolation are fetched from the buffer containing the intensity values in lines 23–30. The if statements make sure, that the image coordinates lie inside of the projection image; for the case where the ray doesn't hit the detector, i.e. the coordinates lie outside the projection image, an intensity value of zero is assumed (line 22). Note that the projection image is linearized, which is why we need the projection image width in the variable width—also made available by the framework via the struct pointer passed to the function—to correctly address data inside the buffer.

The actual bilinear interpolation is performed in lines 32–34. Before the result is written back into the volume, it is weighed according to the inverse-square law. The variable $w$, which holds the homogeneous coordinate $w$, contains an approximation of the distance from x-ray source to the voxel under consideration and can be used to perform the weighting.
for (z=0; z<L; ++z) { // iterate over all voxels in volume
    for (y=0; y<L; ++y) {
        for (x=0; x<L; ++x) {
            // convert from VCS to WCS
            float wx = O + x * MM;
            float wy = O + y * MM;
            float wz = O + z * MM;
            // convert from WCS to ICS
            // dehomogenize
            float ix = u / w;
            float iy = v / w;
            // integer indices to access data in projection image
            int iix = (int)ix;
            int iiy = (int)iy;
            // calculate interpolation weights
            float scalex = ix - iix;
            float scaley = iy - iiy;
            // load four values for bilinear interpolation
            float valbl = 0.0f; float valbr = 0.0f; float valtr = 0.0f; float valtl = 0.0f;
            if (iiy >= 0 && iiy < width && iix >= 0 && iix < height)
                valbl = I[iiy * width + iix];
            if (iiy >= 0 && iiy < width && iix+1 >= 0 && iix+1 < height)
                valbr = I[iiy * width + iix + 1];
            if (iiy+1 >= 0 && iiy+1 < width && iix >= 0 && iix < height)
                valtl = I[(iiy + 1) * width + iix];
            if (iiy+1 >= 0 && iiy+1 < width && iix+1 >= 0 && iix+1 < height)
                valtr = I[(iiy + 1)* width + iix + 1];
            // perform bilinear interpolation
            float valb = (1 - scalex) * valbl + scalex * valbr;
            float valt = (1 - scalex) * valtl + scalex * valtr;
            float val = (1 - scaley) * valb + scaley * valt;
            // add weighted results to voxel
            VOL[z*L*L+y*L+x] += val / (w * w);
        // x-loop
    } // y-loop
} // z-loop

Listing 2.1: Unoptimized reference back projection implementation processing a single projection image.
Chapter 3

Intel 64 and Intel Many Integrated Core Architectures

Treibig et al. have demonstrated that optimizing the FDK reconstruction algorithm on modern CPUs requires a deep architectural knowledge, in particular when it comes to efficient vectorization and parallelization \textsuperscript{[14]}. This Chapter will give a thorough introduction of the hardware design of the two compute devices which were used throughout this thesis.

In addition to the Intel MIC architecture, which is the main focus of this thesis, a typical modern CPU architecture is introduced as well. This CPU system is meant to serve as a reference, which we can compare the approaches and concepts of the Intel MIC to. The reference system used is (at the time of this writing) a state of the art two-socket system, employing two Intel Xeon E5-2680 CPUs clocked at 2.7 GHz, which are based on Intel’s latest Sandy Bridge EP architecture.

We will describe the most important features of both the accelerator as well as the CPU system that are relevant to better understand the optimizations performed in the latter chapters. Apart of the general core design, these include the cache and memory subsystems, as well as vectorization.

In order to examine the aforementioned aspects of the hardware in more detail, we make use of various benchmarks, which are described in Section \textsuperscript{3.1} together with the benchmark environment. We continue with an introduction to the Sandy Bridge microarchitecture in Section \textsuperscript{3.2}, which can then serve as a reference to highlight the major differences in the design of the Intel MIC, which is introduced in Section \textsuperscript{3.3}.

Sections \textsuperscript{3.2} and \textsuperscript{3.3} are based on information found in the \textit{Intel 64 and IA-32 Architectures Optimization Reference Manual} \textsuperscript{[18, 19]} and the \textit{Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual} \textsuperscript{[20]}.

3.1. Preliminaries

3.1.1. Environment

The Sandy Bridge reference CPU system is running a Linux 3.2.0 kernel, while the Intel MIC runs a Linux 2.6.38.8 kernel. The compiler used throughout the thesis is the Intel C Compiler version 13.1.0.
3.1.2. Likwid

The likwid project [21] consists of a collection of command-line tools for the Linux operating system that are intended to assist programmers to develop and run high-performance applications on modern multi- and manycore architectures. Apart from the widely used Intel and AMD processors, likwid also includes support for the Intel MIC architecture. In the following, we will focus on tools that were used in this thesis—likwid-pin, likwid-bench, and likwid-perfctr.

likwid-pin

Whenever a multi-threaded application is run on a Linux kernel, software threads are assigned to hardware threads based on the operating system scheduling strategy. Whenever a context switch occurs, the preempted thread may continue execution on a different hardware thread. This of course becomes a problem when the data the thread was using is not available in the caches of the core it is now running on. To circumvent this issue, threads can be pinned to hardware contexts. This way, even as context switches occur and software threads are preempted from their hardware context, they are guaranteed to resume execution on the same hardware thread they were preempted from. The likwid-pin tool provides an easy and intuitive way to perform this pinning even for complex hardware architectures such as the Intel MIC.

likwid-bench

The likwid-bench tool consists of a framework as well as a set of standard kernels that can be used to evaluate certain properties of hardware platforms, such as cache and memory bandwidths for different memory access patterns. The set of available kernels can easily be extended, enabling rapid prototyping of custom benchmarking kernels.

likwid-perfctr

Most architectures provide a number of hardware performance counters that can record statistics about hardware events such as the number of instructions issued, the number of cache hits, amount of data transferred, etc. These counters can be accessed by likwid-perfctr through the Linux Model Specific Registers (MSR) kernel module and used to profile application performance.

3.1.3. STREAM Benchmark

STREAM [22] is a benchmark designed to measure the sustainable memory bandwidth on a given architectures. It comprises a benchmark interface and a set of simple vector kernels. The Copy kernel is shown as an example in Listing 3.1.

Table 3.1 shows all kernels together with their data intensity and arithmetic intensity.

---

1 Context switches occur even on systems that are exclusively being used for a particular application. Events such as kernel interrupts happen at regular intervals and preempt userland threads on the CPU.
### 3.2. Intel’s Sandy Bridge Microarchitecture

An overview of the CPU reference system layout can be seen in Figure 3.1. The system contains two sockets; each processor package features eight physical cores and provides two SMT hardware threads per core. Each of the superscalar cores is clocked at 2.7 GHz and comes with 32 KiB of Level One (L1) and 256 KiB of Level Two (L2) cache; the cores are connected together via the Last Level Cache (LLC) ring interconnect. The microarchitecture also features Intel’s latest set of vector extensions, called Advanced Vector Extensions (AVX), which supports up to eight single- or four double-precision floating-point operations per instruction.

---

#### Listing 3.1: Copy kernel used in the STREAM benchmark.

```c
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
c[j] = a[j];
```

#### Table 3.1.: The implementations of the entire set of kernels that come with the STREAM benchmark along with the bytes per iteration and floating-point operations per iterations values for each kernel.

<table>
<thead>
<tr>
<th>Name</th>
<th>Kernel</th>
<th>bytes/iter</th>
<th>Flops/iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>c[j] = a[j];</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Scale</td>
<td>b[j] = scalar*c[j];</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>Add</td>
<td>c[j] = a[j]+b[j];</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>Stream</td>
<td>a[j] = b[j]+scalar*c[j];</td>
<td>24</td>
<td>2</td>
</tr>
</tbody>
</table>

The STREAM binary we used in this thesis was compiled using the Intel Compiler with the `-O3`, `-openmp`, and `-opt-streaming-stores always` compiler flags.

#### 3.1.4. Plots

To compensate for variation in runtimes the plots presented in this thesis show results that were obtained by averaging several sample runs of a benchmark. The plots are boxplots in which the data point represents the mean value obtained by averaging over all results. The boxes indicate the 90th percentile, i.e. the upper box bound indicating that 90% of all values lied below this threshold, the lower box bound indicating that 90% of all values lied above this threshold. The whiskers indicate the minimum and maximum values obtained over all runs.

Because different benchmarks take different times to execute, the number of samples performed for each benchmark varies (due to time-constraints on the hardware) and is therefore given with each plot.
3. Intel 64 and Intel Many Integrated Core Architectures

Figure 3.1.: General layout of the two-socket Sandy Bridge EP system. Each processor package features eight physical cores and the packages are connected via two QuickPath Interconnect (QPI) links. The four 64-bit DDR3-1600 memory channels offer a maximum theoretical bandwidth of 51.2 GiB/s per processor package.

In the following, we will discuss the most important parts of the system in more detail.

3.2.1. Core Pipeline

The core pipeline, shown in Figure 3.2, can be separated into two major components:

- The in-order front end, which is responsible for fetching instructions from the Level 1 instruction cache and decoding them to micro-operations—also known as micro-ops or uops.

- The out-of-order, superscalar execution engine, which dispatches up to six of the previously decoded micro-ops to the execution units each clock cycle. Instead of simply issuing uops sequentially, which could cause stalls due to dependencies, the execution engine can reorder micro-ops to so-called “dataflow” order. This way, under the premise that data dependencies are honored—thus ensuring the semantic correctness of the program—the core can reorder instructions at runtime to maximize throughput.

To better understand the interactions between the two different parts of the core and their respective inner workings, we will examine the way instructions flow through the pipeline in more detail.

The part of the front end that is responsible for getting instructions into the core is called Instruction Fetcher (IF). If there is non-sequential execution, the IF relies on the Branch Prediction Unit (BPU) to predict which data the core is probably going to require next. Using several heuristics, the BPU determines the path that the program is most likely to take; the instruction fetcher then gets the designated next block of code to execute.
3.2. Intel’s Sandy Bridge Microarchitecture

Figure 3.2.: Core pipeline of the Sandy Bridge microarchitecture. The upper half shows the in-order front end, which tries to deliver instructions to out-of-order execution engine as fast as possible. The bottom half shows this execution engine, which tries to maximize instruction throughput by reordering instructions to dataflow order. Both parts of the pipeline communicate with the memory system through the caches: The frontend is loading instructions from the L1 instruction cache while the execution engine is loading the instruction operands from and writing the results back to the L1 data cache. (Image based on Figure 2-3 of the Intel 64 and IA-32 Architectures Optimization Reference Manual [19])
The IF first searches the decoded instruction cache (also referred to as L0 cache) for the code. This cache holds uops that have already been decoded in previous cycles. If the instructions are not found in the L0 cache, the IF has to bring in the data from the L1 cache and decode the instructions using the legacy decode pipeline. If there is a L1 miss, the instructions have to be brought into the L1 cache from the cache/memory hierarchy, i.e. the unified L2 cache, the last-level cache (LLC), or memory. Once the data is in the L1 instruction cache, it can be brought into the predecode unit at 16 byte/cycle.

The predecode unit determines the instructions’ lengths based on the Length Changing Prefixes (LCPs), which may indicate larger operand sizes, i.e. instructions with immediate operands larger that the default size of one byte (e.g. word or doubleword immediates); address size override, i.e. instructions with the ModR/M after the opcode, indicating the addressing mode to be used in the instruction; or prefixes (e.g. REX, VEX).

In the next step, the instructions are placed in the instruction queue, from where they are fed into the instruction decode units. There is a total of five decoding units, some of which serve special purposes: Apart from three simple decoders, each of which can decode one instruction to micro-op per cycle, there is a complex decoder, which can covert more complex instructions into up to four uops; another special unit, called micro-sequencer ROM, is dedicated to a set of even more complex instructions, and can decode up to four uops per cycle. Each decoded instruction is passed to the decoded instruction cache as well as the decoded instruction queue.

Once the uops have been placed in the decoded instruction queue, the other major component of the pipeline takes over. This queue can thus be seen as link between the in-order front end and the out-of-order execution engine of the core. Apart from this, the queue serves two other purposes:

- It helps to hide bubbles, which may be introduced by the front end (e.g. L0 miss, L1 miss, branch misprediction), by trying to ensure that four uops are delivered for execution per clock cycle.
- It also provides post-decode functionality for several CISC-instructions like arithmetic operations that use a memory source or mov instructions that use indexed addressing. In order to reduce bandwidth inside the front end, complex instructions such as these are decoded into a single uop, which means they are also present as single uops in the L0 cache. The decoded micro-instruction queue uses a process called unlamination to split them up into several uops for execution. An example could be an addition, which loads a floating-point value from memory using indexed addressing and accumulates the result in a register: vaddsd ymm0, ymm0, [rdi+ecx*8]—this would be split up into a load (mov tmp0, [rdi+ecx*8]) and the actual addition (vaddsd ymm0, ymm0, tmp0); the load in turn is split up into an address generation (lea tmp1, [rdi+ecx*8]) and mov instruction (mov tmp0, [tmp1]).

After unlamination, the out-of-order execution engine takes over. This engine contains three major components: The renamer, which moves micro-ops from the front end, i.e. the

---

2 The registers tmp0 and tmp1 are microarchitectural registers not visible to the programmer. The names were chosen arbitrarily.
decoded micro-instruction queue, to the execution core; the scheduler, which queues uops until all their source operands are available and then dispatches them to the execution units; and the retirement unit, which actually executes the uops on the assigned execution units in the order assigned by the scheduler.

The renamer, together with the decoded micro-op instruction queue, form the bridge from the front-end to the execution engine. It can move up to four uops per cycle from the queue into the engine, in the process renaming architectural sources/destinations (e.g. \texttt{eax, r15d, xmm0, ymm11}) of the uops to microarchitectural sources/destinations (hardwired register names within the register file not visible to the programmer). The renamer also selects the appropriate issue ports for a micro-op.

The scheduler is in charge of dispatching uops to the issue ports. To maximize throughput, the scheduler tries to detect dependency chains in uops and eliminate fake dependencies. Whenever a uop has to be queued, because one of its sources is not available, the scheduler iterates through all dependency chains until it finds a micro-op whose sources are available.

Finally, the retirement units executes the previously scheduled uops on the corresponding issue port in FIFO order.

The execution core, made up by the six issue ports, is superscalar in design, i.e. it can execute uops on multiple issue ports in parallel in each cycle. This execution of uops in parallel is also known as Instruction Level Parallelism (ILP). The memory control portion of the core connects the L1 data cache to the issue ports; of course, it can also directly feed back results from issue ports to the various execution stacks, enabling the reuse of generated results as inputs for successive uops. The same reasoning also applies to loads and stores: A store to a memory address can directly be fed back into an issue port to decrease the latency a subsequent uop would experience when it would naively load the data from L1. To avoid delays, the memory control implements separate buses for each domain. This means, that while a stall occurs on the bus connected to issue port 0 if we retire a \texttt{vaddps} instruction (which has a latency of 3 cycles) on that port, and retire a \texttt{vblendps} instruction (which only has a latency of one cycle) two cycles later we get a bus conflict, because both instructions finish at the same cycle and both use the SIMD domain; however, issuing a \texttt{vaddps} instruction on the same port followed by an \texttt{add} instruction (which has one cycle latency) two cycles later we will not cause a conflict, because the former uses the SIMD floating-point domain, while the latter uses the integer domain, and each domain has its own bus.

### 3.2.2. Cache Organization, Core Interconnect, and Memory

Each core of a processor package contains a 32 KiB first level instruction cache (L1 i-cache), a 32 KiB first level data cache (L1 d-cache), and a 256 KiB unified, i.e. shared by instructions and data, second level cache (L2 cache). All cores of a package are connected to a shared third level cache (L3 cache)—also known as last level cache (LLC)—using a ring topology. The size of the LLC varies with the number of cores, with each core typically contributing 2–
3. Intel 64 and Intel Many Integrated Core Architectures

3 MiB to the LLC. In the Xeon E5-2680 CPU each core contributes a total of 2.5 MiB, yielding a total of 20 MiB per package.

To increase efficiency, all data transfers between memory and cache, as well as caches themselves, occur in 64 byte blocks; these blocks are usually referred to as Cache Lines (CLs). Cache coherency is maintained across all caches using the MESI protocol \cite{18}.

When data is written—we assume a “standard” store, not a non-temporal store\footnote{Non-temporal stores can be used when writing all 64 bytes in a CL. Instead of writing the data into the cache, special streaming buffers are used, which, when a CL is fully filled, update the main memory directly without going through the cache hierarchy.}—the cache logic makes sure that the respective CL already exists in the core’s L1 d-cache in either the exclusive or modified state. In case the CL is not already present in the cache a write-allocate occurs, i.e. the CL is fetched from further down the memory hierarchy (L2, L3, or eventually memory); once the CL has been fetched into the first level data cache the store can occur.

In the following, we have a closer look at the design and resulting implications of the three levels in the cache hierarchy.

**First Level Caches**

The L1 instruction and data caches both have a latency of 4 clock cycles and are 8-way associative. The L1 d-cache has a bandwidth of 48 bytes per cycle to the core—it can receive 16 bytes of data from the memory control of the core while at the same time providing 32 bytes of data to it. This is equal to one AVX load and half an AVX store per cycle (cf. Section \ref{3.2.3}). Obviously, data will not be written back into the L1 i-cache, which is why the link to the predecode unit is unidirectional. The L1 i-cache provides 16 bytes of instructions to the predecode unit per clock cycle.

Apart from explicit loads, data can be moved into the L1 d-cache using software prefetching, hardware prefetching, or a combination of both.

Software prefetching is done using one of the Streaming SIMD Extension (SSE) prefetch instructions, `prefetch0`, which will fetch the data into the L1 d-cache (implicitly fetching it into L2 and L3, if necessary); however, this prefetch instructions should be taken with caution, as it is merely a hint that the hardware can choose to ignore.

On the hardware side, both the streaming and the stride prefetcher are responsible for prefetching data into the L1 d-cache. The streaming prefetcher is triggered by continuous memory access patterns and will automatically fetch the next cache line in the stream. The stride prefetcher keeps track of the addresses used in individual load instructions; if loads are detected with regular strides a prefetch is send to the next address, which is the sum of the current address and the stride.

**Second Level Cache**

The L2 cache has a latency of 12 clock cycles, is 8-way associative, and can serve up to 32 bytes per clock cycle. While the L1 d-cache can communicate bidirectionally in each clock cycle—e.g. receive 16 byte while sending 32 byte to the memory control—the L2 cache can either...
3.2. Intel’s Sandy Bridge Microarchitecture

send 32 byte or receive 32 byte from the L1 d-cache per clock cycle.

Analogous to the L1 d-cache, L2 cache data is prefetched using either software prefetches, hardware prefetchers, or a combination of both.

The `prefetcht1` SSE instruction will prefetch a CL into the L2 cache (implicitly prefetching into L3, if necessary).

The hardware prefetchers that are responsible to fetch data into the L2 cache are called spatial prefetcher and streamer. The spatial prefetcher tries to supplement each CL fetched into the L2 cache with the pair line that completes it to a 128 byte aligned chunk, i.e. fetching the CL directly before or after a certain CL. The streamer continuously monitors the read requests\(^5\) from the L1 cache for ascending and descending sequences of addresses; whenever such a stream is detected, the anticipated cache lines are prefetched.

Last Level Cache and Core Interconnect

According to Intel, the associativity of the LLC can vary between different CPU models implementing the Sandy Bridge architecture. The CPU used in this thesis used a L3 cache which is 16-way associative. Due to the sliced design of the shared cache, the latency can vary from 26 to 60 cycles, depending on where the data has to be fetched from.

If the CL containing the data has not been modified, the data can be fetched in 26–31 cycles, depending on the distance to the slice the CL has to be fetched from. However, if the CL is marked as exclusive in a remote slice, the procedure is more complex: Depending on whether the CL has actually been modified at the time of the request, either a clean hit (unmodified) or dirty hit (modified) occurs. In this case the latency is 43 clock cycles for a clean hit and 60 cycles for a dirty hit—the additional penalty in latency for the dirty hit is caused by the CL having to be pulled from the cores L1/L2 cache; if the data isn’t actually modified, the CL in the LLC is still valid and can be serviced into the requesting core’s cache from there.

It is worth noting that a single core can not only get data from a remote core’s slice, but also place data into a remote slice. In fact, this is the default behavior. Whenever a core fetches data from memory, the LLC control logic uses a hash function to determine the slice in which the data will be placed. The motivation for this reasoning is to distribute the data uniformly between the slices. Apart from increasing scalability across cores (see Figure 3.3) this has another highly positive implication, namely, that even sequential programs that run on only one core can benefit from the multi-core CPU design by being able to use the cache slices of all cores, resulting in an effective L3 cache size of 20 MiB, independent of the actual number of cores used (see Figure 3.4).

It becomes clear that by enabling individual cores to both read from and write to remote core’s slices, the LLC serves as interconnect between all cores of package.

Memory Subsystem

Each of the two processor packages is connected to its main memory via four 64-bit DDR3 memory channels. Both packages can also access the memory of the other package over the\(^5\) These include implicit loads, software prefetches, and hardware prefetches.
3. Intel 64 and Intel Many Integrated Core Architectures

Figure 3.3: L3 cache performance of one processor package. Results obtained with likwid-bench using 100 iterations of the copy benchmark, 10 MiB of data, and varying core count.

Figure 3.4: L3 cache and memory performance of a single core. Results obtained with likwid-bench using 100 iterations of the copy benchmark and varying memory size.
Intel's Sandy Bridge Microarchitecture

QPI links that connect the processors together. However, going through the QPI links will increase latency when accessing remote memory which is why this design is also referred to as a Non-Uniform Memory Access (NUMA) topology. One package supports a maximum DDR3 clock frequency of 1.6 GHz, which yields a theoretical peak memory bandwidth of 51.2 GiB/s per NUMA-domain (i.e. processor package and its directly attached memory), or 102.4 GiB/s per node.

Due to limitations in memory controller efficiency it is impossible to achieve the theoretical bandwidth in practice. Typically, the Copy kernel from the STREAM benchmark using streaming stores is used to get a realistic estimate for the general bandwidth that can be expected. In addition to the Copy kernel, we also show the results of an additional Update kernel (see Listing 3.2), which we added to the original STREAM benchmark. This Update kernel was designed to resemble the data access pattern used in the actual CT reconstruction implementation; it should thus give the best possible estimation for the bandwidth we can anticipate for our application.

```
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
a[j] = scalar + a[j];
```

Listing 3.2: Update kernel we used to benchmark the memory bandwidth.

While, at first glance, this code might look similar to the Copy kernel glance, the bandwidth achieved by the Update kernel is very different. While the Copy kernels deals with two distinct streams from memory (i.e. reading from array c[] and writing to array a[], cf. Listing 3.1), the Update kernel requires only one stream (that is both read and written). Even with non-temporal stores, the performance of the Copy kernel is much worse than that of the Update kernel, as can be seen in Figure 3.5.

The achieved performance for the Update kernel of roughly 44 GiB/s corresponds to about 86% of the theoretical peak performance of one socket (51.2 GiB/s).

3.2.3. Advanced Vector Extensions

The Sandy Bridge microarchitecture introduced Intel's latest set of vector extensions, called AVX. In addition to doubling the SIMD width of the 16 vector register from SSE's 128 bit to 256 bit, the AVX instructions also come with a new instruction coding called VEX coding, which extends the existing x86-64 instruction set architecture. AVX adds arithmetic instructions for variants of add, subtract, multiply, divide, square root, compare, min, max, and round on single- and double-precision floating-point data [I8]. At the same clock rate, the new AVX instructions offer twice the peak Flop/s rate of Intel's previous microarchitecture. Depending on the floating-point precision, a single instruction can now be simultaneously applied to eight (single precision) or four (double precision) values (see Figure 3.6).

These extensions can be used either directly in assembly or via intrinsics in high-level languages such as C or Fortran, or automatically by the compiler. While the C standard imposes
Figure 3.5: Results of the STREAM Copy and Update benchmarks with \texttt{STREAM\_ARRAY\_SIZE} the original value of 10 million elements. The bandwidth shown is the actual bandwidth, because there are no implicit write-allocates: The Copy kernel uses non-temporal stores to avoid write-allocates and there can be no write-misses in the Update kernel.

Figure 3.6: Mapping of different data types to vector registers. The upper left mapping shows how eight 32 bit single-precision floating-point values fit into a single 256 bit vector register; to maintain compatibility with the outdated SSE vector extensions, the lower 128 bits of the new 256 bit vector registers are used whenever old SSE instructions are used. The right part of the image shows the mapping for double-precision floating-point numbers.
several restrictions as to where the compiler can use vectorization, there exist several compiler directives which allow programmers to harness the SIMD capabilities of modern CPUs through the use of compiler hints.

```c
void add(float *A, float *B, float *C, int N) {
    #pragma vector aligned
    for (i=0; i<N; ++i)
        C[i] = A[i] + B[i];
}
```

Listing 3.3: Loop code that adds single-precision floating-point values.

Consider the C function shown in Listing 3.3. Despite the simple loop structure, a compiler would not be allowed to vectorize this loop due to the fact that the C language standard allows for pointer aliasing: Consistent with the C language standard, \(B[i]\) could very well point to the same element as \(A[i-1]\), thereby introducing a loop dependency, making it impossible to vectorize the code. We can, however, use a compiler option—e.g. `-fno-alias` for the Intel compiler—to tell the compiler that our code will not use pointer aliasing at all; then, it can safely vectorize the loop. The `#pragma vector aligned` directive before the loop is another hint for the compiler. By guaranteeing the compiler that our memory is aligned, it can safely use the aligned load instruction (`vmovaps`) and not fall back to default unaligned load counterpart (`vmovups`).

With vectorization in place, the function above can be sped up quite extensively with regard to instructions required to complete the loop: In the unvectorized version, each loop is mapped exclusively to scalar operations. The addition of the first eight elements would require a total of 24 `mov` instructions (8 loads for \(A[i] \ldots A[i+7]\), 8 loads for \(B[i] \ldots B[i+7]\), and 8 stores for \(C[i] \ldots C[i+7]\)), as well as 8 `add` instructions. The vectorized code can work on eight elements in parallel, i.e. a total of four instructions can process the first eight elements: A `vmovaps` will move 256 bits, corresponding to eight 32-bit single-precision floating-point numbers, from \(A[]\) into a vector register; a second `vmovaps` will load eight `float`s from \(B[]\); the `vaddps` will add the contents of the two vector registers (cf. Fig. 3.7); and, finally, another `vmovaps` will move the result into \(C[]\). Assuming \(N\) is a multiple of eight, the code is eight times more efficient instruction-wise.

To put things into perspective, we can also change our point-of-view: By not using vectorization, programmers sacrifice up to \(7/8 = 87.5\%\) (resp. \(3/4 = 75\%\) when dealing with double-precision) of the performance that today’s CPUs offer.

Another novelty of AVX is the use of up to four operands, made possible by the previously mentioned VEX coding scheme. One benefit of this is that operations can be made non-destructive with respect to the source operands: The old two-operand SSE instruction `addps xmm1, xmm2`, which adds the values stored in `xmm1` and `xmm2` would destroy the original values in the `xmm1` when storing the result to this register; the new VEX coding, which allows up to four operands, resolves this problem by simply storing the result to another register.

\(^6\)Depending on the actual microarchitecture, the unaligned load can be much slower than the aligned load, because it has to account for the possibility that the data to be fetched is distributed across multiple CLs.
3. Intel 64 and Intel Many Integrated Core Architectures

Figure 3.7: Illustration of vector addition: The vector registers $\text{ymm}X$ and $\text{ymm}Y$ contain the first eight single-precision floating-point numbers stored at the memory addresses $\text{A}$ and $\text{B}$. The vector addition will perform component-wise additions of the contents of the registers and store the results in $\text{ymm}Z$.

3.3. Intel’s Many Integrated Core Microarchitecture

An overview of the Xeon Phi 5110P is provided in Figure 3.8. The main components making up the accelerator are 60 cores arranged on a high bandwidth ring interconnect known as Core-Ring Interconnect (CRI); interlaced with the ring is a total of eight GDDR5 memory controllers to connect the cores to main memory as well as PCIe logic that communicates with the host system.

The cores are based on a modified version of P54C design, used in the original Pentium released in 1995. Each core is clocked at 1.05 GHz and is a fully functional, in-order core, which supports fetch and decode instructions from four hardware thread execution contexts—double the amount than previously seen with in the Sandy Bridge architecture. The cores are superscalar, featuring a scalar pipeline (V-pipe) and a vector pipeline (U-pipe). The Vector Processing Unit (VPU) that is connected to the U-pipe features a total of 32 512-bit vector registers and is capable of fused-multiply add operations, yielding a total of 16 DP (32 SP) Flops per cycle. The VPU is accessed using a new set of vector extensions, called Initial Many Core Instructions (IMCI). One of the novelties of the IMCI is the support for vector scatter and gather operations, as well as the already mentioned support for fused-multiply add operations.

In addition to the execution pipelines, each core comes with 32 KiB of L1 cache, the same amount as recent traditional Intel CPUs. The L2 cache size is 512 KiB and kept fully coherent.
3.3. Intel’s Many Integrated Core Microarchitecture

by a globally distributed Tag Directory (TD) (discussed in detail in Section 3.3.2).

In the following, we discuss the most important parts of the accelerator in more detail.

![Diagram of the Xeon Phi 5110P accelerator](image)

Figure 3.8: General layout of the Xeon Phi 5110P accelerator. For simplicity, only 8 of the total 60 cores and 4 of the total 8 GDDR5 memory controllers are shown.

3.3.1. Core Pipeline

The Intel MIC’s core pipeline, which can be seen in Figure 3.9, has a much simpler design than the Sandy Bridge pipeline. The entire core is in-order, lacking all of the necessary logic to manage out-of-order execution, making the individual cores less complex than their traditional CPU counterparts.

A core can execute two instructions per clock cycle: One on the V-pipe, which executes scalar instructions, prefetches, loads, and stores; and one on the U-pipe, which can only execute vector instructions. The scalar V-pipe of the core provides a similar execution environment to that found in traditional Intel 64 architectures. The U-pipe is connected to the 512 bit wide vector processing unit (VPU), which is used for vector instructions. Only IMCI are supported on the VPU—legacy support for neither MMX, SSE, nor AVX is provided.

Instructions enter the core the same way as in the Sandy Bridge pipeline. A BPU provides the next most likely path for each of the hardware threads and the IF fetches 16 bytes of instructions for any one of the four hardware thread per clock cycle. The fetched instructions are separated into two 8-byte bundles, each bundle ideally containing two instructions—
Figure 3.9: Core pipeline of the Intel MIC architecture: The instruction fetcher, together with the branch prediction unit, gets the data into the prefetch buffers of the four hardware contexts. The decode unit decodes the bundles from these buffers and the picker function selects the hardware context to run on the V- and U-pipe at a given clock cycle. The left part of the image shows how a core can access the ring interconnect via its own private core-ring interconnect.
3.3. Intel’s Many Integrated Core Microarchitecture

one for the V-pipe and one for the U-pipe. With the instruction bundles in place, the Picker Function (PF) will choose the next hardware thread that is ready to execute in a round-robin manner and send its bundle to the instruction decoder.

The decode unit is shared by all hardware threads and is a pipelined two-cycle unit to increase throughput. This means it takes the decode unit two cycles to decode one instruction bundle (i.e. one uop for the U- and one uop for the V-pipe); by its pipelined design, the unit can, however, deliver decoded bundles to different hardware threads each cycle. A major implication of this design is that no hardware thread can execute instructions in two successive cycles: If a hardware thread executed instructions at cycle \( n \), then only the other hardware threads can execute at cycle \( n + 1 \), because decoding the next instruction for the original thread will take another cycle. As a consequence, at least two hardware threads must be run on each core to achieve peak performance; using only one thread per core will in the best case result in 50% of peak performance.

Unfortunately, there is very few information available about the pipeline from Intel. While one would expect there exist similar facilities to forward memory stores directly back into the execution units, the Intel Xeon Phi System Software Developer’s Guide explicitly states a store-to-load penalty of 12 clock cycles for the L1 cache \([23]\).

3.3.2. Cache Organization, Core Interconnect, and Memory

Most of Intel’s cache concepts were adopted into the MIC: The cache line size is 64 bytes and cache coherency is is implemented across all caches using the MESI protocol. Each core includes a 32 KiB first level instruction cache (L1 i-cache), a 32 KiB first level data cache (L1 d-cache), and a unified 512 KiB second level cache (L2 cache). As there is no third level cache on the Intel MIC, the L2 cache acts as a shared cache; however, as will be demonstrated later, this shared cache behaves rather differently in certain respects than the shared LLC of the Sandy Bridge microarchitecture.

Again, we will have a closer look at the different cache levels, comparing them to the Sandy Bridge architecture and extracting the most important differences.

First Level Caches

The L1 d-cache is 8-way associative and has a 1 cycle latency for scalar loads and a 3 cycle latency for vector loads. Its bandwidth has been increased to 64 bytes per cycle, which corresponds exactly to the vector register width of 512 bits.

In contrast to the Sandy Bridge CPU, which contained two hardware prefetching units for the L1 d-cache (streaming prefetcher and stride prefetcher), there exist no hardware prefetchers for the first level cache on the Intel MIC. As a consequence, the compiler/programmer has to make heavy use of software prefetching instructions—which are available in various flavors (cf. Table 3.2)—to make sure data is present in the caches whenever needed.

---

7 Obviously, if two instructions meant to run on the U- and V-pipe do not conform to these size constraints they cannot be combined into the same bundle. Since only one bundle can be executed at a time, this means that either the U- or V-pipe will be unused in at least one clock cycle.
Table 3.2.: Available scalar prefetch instructions for the Intel MIC. Apart from standard prefetches into the L1 and L2 caches (vprefetch0, vprefetch1), there exist also variants that prefetch data into what Intel refers to the L1/L2 non-temporal cache (vprefetchnta, vprefetch2). Whether these caches are actually a portion of the normal L1/L2 caches or separate, smaller caches does not become clear from the documentation. Data prefetched into these non-temporal caches is fetched into the \( n \)th way of the cache, where \( n \) is the context id of the prefetching hardware thread—e.g. if hardware context 0 was issuing the prefetch, the data is brought into way 0 of the cache; if hardware context 1 issued the prefetch, data is brought into way 1; and so on—and made MRU, i.e. the most recently used data will be replaced first. Prefetches can also indicate the requesting CL be brought into the cache for writing, i.e. in the exclusive state of the MESI protocol (vprefetch*).
Second Level Cache

The L2 cache is 8-way associative and has a latency of 11 clock cycles—somewhat similar to the 12 cycles of Sandy Bridge. While the size of the L1 d-cache was not increased, the L2 cache size was doubled to 512 KiB to compensate for twice the number of hardware threads.

Although all hardware prefetches for the L1 d-cache were removed, the L2 cache could at least retain one of the two prefilters that were available on Sandy Bridge, namely, the streaming prefﬁter. However, the prefﬁter is far less powerful than its Sandy Bridge counterpart. In order for it to detect strides, two successive memory accesses may only be up to 2 CLs apart; also, it can only monitor up to 16 streams, which have to be shared between the four hardware threads.

A peculiarity of the level 2 caches is how they work together to form the “shared” last level cache. As we have seen before, the LLC slices in the Sandy Bridge microarchitecture were only physically associated with a core, and not considered core-private in the sense that only the associated core can place data in the slice; rather, each core would distribute CLs it fetches from memory into the slice that a hash function determines based on a given CL’s address.

The Intel MIC does not implement this strategy. Whenever a core reads data from memory, the corresponding CL is always placed in the L2 cache that is associated with the core. As a direct consequence, the maximum cache size $s_{\text{max}}$ is a function of the number of cores $n$ that are used: $s_{\text{max}} = n \cdot 512$ KiB. This effect can be observed in Figures 3.10 and 3.11.

![Figure 3.10: L2 cache and memory performance of a single Intel MIC core. Results obtained with likwid-bench using 100 iterations of the copy benchmark and varying memory size.](image-url)
Figure 3.10 shows the achieved bandwidth in relation to the data size used in a likwid-bench *copy* benchmark when using a single core. While the Sandy Bridge CPU allowed one core to distribute the CLs into all slices of the LLC and could thus make use of the whole processor package’s LLC (cf. Figure 3.4), a single Intel MIC core will fall back to the main memory bandwidth once the data is too large for its private L2 cache (512 KiB), because it can not place data in remote core’s L2 caches.

![Graph showing bandwidth vs core count](image)

**Figure 3.11:** Memory/L2 cache performance of the entire Intel MIC chip (60 cores). Results obtained with likwid-bench using 100 iterations of the *copy* benchmark, 15 MiB of data, and varying core count.

While on Sandy Bridge, the ring interconnect scales with the number of cores (even in the worst case that data for each core was located in another core’s slice), a Intel MIC core will always find the data it has previously fetched from memory in its own L2 cache. This scalability can be observed in Figure 3.11. In this run, we vary the number of used cores with a fixed work set of 15 MiB for a hundred iterations. The increase in bandwidth up to around 25 cores is that of memory bandwidth, because the aggregate L2 cache size is too small. Actually, we would expect the attained bandwidth to increase above main memory bandwidth only after we use 30 cores. The reason why the increase is happening before can be traced back to the cache replacement strategy, i.e. the strategy which dictates what CL to substitute or evict once the cache is full. The L2 cache on the Intel MIC uses a pseudo-LRU (Least Recently Used) in favor of a strict LRU strategy. While we do not know the exact pseudo-LRU implementation, the results indicate that there appears to be some sort of (pseudo-)randomness involved in the choice of the CL to replace. This way, there is a certain probability that a CL will still be in the cache, even after the L2 cache has been filled with more than 512 KiB of data. Incidentally, the L3 cache of Sandy Bridge uses a pseudo-LRU replacement policy as well, which is why we...
3.3. Intel’s Many Integrated Core Microarchitecture

small to hold one core’s share of the total data, implying the data has to be fetched from main memory in each iteration. Only after we use about 25–30 cores, the aggregate L2 cache size is large enough to hold the entire data; at this point, we find an increase in slope of the graph, because the data is no longer coming from main memory but from L2 cache. Of course, the increase in L2 cache bandwidth scales linear with the number of cores, because there is no communication between the caches of different cores involved and each core only reads data from its own cache segment. In addition to the cache scalability we find that the use of SMT is mandatory for good in-cache performance.

Core Interconnect

The L2 cache is part of the core-ring interface block. As we have seen before (cf. Fig. 3.8), this block also contains a core’s portion of the globally distributed tag directory and is the interface to the bidirectional ring interconnect. We have seen previously that a core can not place data into a remote core’s L2 cache. It can, however, read data from remote caches—this is done over the core interconnect.

This interconnect is implemented as a bidirectional ring, with each direction comprising of three dedicated channels: The data block ring, which is responsible for moving actual data at a rate of 64 byte per cycle; the address ring, used to send and receive commands requesting memory; and the acknowledgment ring, used to send flow control and coherence messages.

Whenever a core’s access to its private L2 cache results in a miss, the core queries the globally distributed tag directory in order to find out if another core’s cache contains the data it is requesting. This request is sent over the address ring. In order to increase scalability, a hashing function is used to uniformly distribute tags into the tag directories. The tag directory will contain information on whether the data is available in a core, and, if it is, which core holds the data. The tag directory will forward the request to the appropriate core’s cache on the address ring and the core receiving the request will forward the CL over the data ring to the requesting core. If the tag directory does not contain an entry for the requested CL it will initiate a request for the CL to one of the memory controllers. It is worth noting that the TD will not turn to the nearest memory controller to fulfill the request, but choose a particular controller depending on the result of a hashing function applied to the CL’s tag; again, this strategy aims to distribute load among all memory controllers.

Once the data is received by the requesting core, it is replicated into its own L2 cache and execution can resume. Note that this fact of data replication clearly contradicts Intel’s remarks about the property of the “shared” L2 cache:

> How all cores work together to make a large, shared, L2 global cache (up to 31 MiB) may not be clear at first glance. [...] (Source: [23])

The reason it “may not be clear” is because it is not possible. While the aggregate cache size certainly is 30 MiB (60 × 512 KiB), the data in the caches can not be shared without decreasing the effective cache size. Whenever a core pulls data from another core’s cache, it didn’t see a sharp drop in Figure 3.4 when using more than 20 MiB of data but more of a smooth decay in bandwidth.
replaces some of the data in its own core, thereby reducing the amount of possibly “shared
data.” If all caches access the same shared data, this results in an effective cache size of only
512 KiB.

Memory Subsystem

The Intel MIC chip comes with a total of eight GDDR5 memory controllers, which are
interleaved symmetrically around the bidirectional ring; each memory controller features
two 32 bit channels and is clocked at 5 GHz, yielding a theoretical peak performance of
320 GiB/s—more than three times than the two-socket Sandy Bridge EP reference system.

To measure the attainable memory bandwidth the Update kernel is used again (cf. Listing
3.2). As we can see in Figure 3.12, the Update kernel achieves a much better performance
than the Copy kernel, which is similar to the situation we found on the Sandy Bridge archi-
tecture (cf. Section 3.2.2).

![Graph showing memory bandwidth vs core count]

Figure 3.12.: Results of the STREAM Copy and Update benchmarks with STREAM_ARRAY_SIZE
the original value of 10 million elements. The bandwidth shown is the actual
bandwidth, because there are no implicit write-allocates: The Copy kernel uses
non-temporal stores to avoid write-allocates and there can be no write-misses in
the Update kernel.

We find that peak memory performance can only be achieved by employing SMT; interest-
estingly, the bandwidth for the Update kernel of 160–170 GiB/s of 320 GiB/s corresponds to
only 50–55% of the theoretical peak performance of 320 GiB/s, implying either a lower mem-
ory controller efficiency than on Sandy Bridge, or a scalability problem (we can see that the gradient of the graph is steeper for e.g. 10–20 cores than it is for 50–60 cores).

3.3.3. Initial Many Core Instructions

The Many Core Instructions (IMCI) is the set of instructions that is used by the vector processing unit associated with the U-pipe. It uses the same VEX coding scheme as Sandy Bridge’s AVX (cf. Section 3.2.3). Both the number of vector registers as well as the vector register width has been doubled: While AVX provides a total of 16 vector registers, each 256 bits wide, IMCI offers 32 register, each now 512 bits wide. This means a single vector operation can now process up to 16 single-precision or 8 double-precision floating point numbers (see Figure 3.13). With the addition of a fused-multiply-add operation to the set of vector instructions this means that a single core can perform up to 32 single-precision or 16 double-precision floating-point operations per cycle.

In addition to increasing the register count and width, IMCI also introduced eight vector mask registers, which can be used to optionally mask out SIMD lanes in vector instructions; this means that a vector operation is performed only selectively on some of the elements in a vector register (see Figure 3.14). The vector mask registers are 16 bits wide in order to address the up to 16 SIMD lanes—e.g. when using single-precision values; if the operation used together with the vector mask register uses double-precision only the lower 8 bits of the vector mask register are used to mask out respective the SIMD lanes.

In order to optimize our code to achieve the maximum theoretical performance it is important to know all instruction latencies when working with in-order hardware. On Sandy Bridge, the out-of-order hardware supports hiding potential latencies in our code by reordering instructions to maximize throughput. On in-order architectures, it is left to the compiler or programmer to order the instructions in such a way that no unnecessary waiting for dependencies occurs. Intel mentions that “most” of the vector instructions have a latency of four clock cycles. This means, that we can simply work around all dependency issues by using all four hardware threads: Whenever thread 0 issues an instruction with four cycles latency,

---

Figure 3.13: The Figure shows the mapping of different data types to the 512 bit vector registers. Each of the 32 vector registers can hold up to sixteen single-precision (upper part) or eight double-precision (lower part) floating-point values.

In fact, the first of the vector mask registers is a special purpose register that has a mask of all 1 bits. This means that there are only 7 freely usable vector mask registers.
3. Intel 64 and Intel Many Integrated Core Architectures

by the time thread 0 will regain control, its instruction will have finished. In the mean time all other hardware threads will be scheduled. However, this disregard for latencies holds only true if all and not most instructions have a latency of four cycles. In order to determine whether instruction latencies would be a problem we had to check the latencies for all the instructions used in our reconstruction kernel. Unfortunately, there is no list with latency and throughput values for IMCI instructions, which is why a way to reliably measure instruction latencies had to be devised. The resulting assembly code can be seen in Listing 3.4. This template can easily be used to measure various instruction latencies by simple changing the value of the INSTR macro to the desired assembly instruction. Measuring the walltime of the loop, one can infer the instruction latency by converting the runtime from seconds into clock cycles using the processor’s frequency and then dividing by the total number of instructions. The results for all instructions used in the reconstruction kernel can be seen in Table 3.3.

The obtained results were quite promising. Almost all of the instructions used in our code adhere to the advertised four cycles latency. Only the calculation of the reciprocal using vrcp23ps required special scrutiny to avoid unnecessary stalls in execution.

3.3.4. Vector Gather Operation

In the FDK algorithm, a lot of data has to be loaded form different offsets inside the projection image. The Intel MIC offers a vector gather operation that enables filling of vector registers with scattered data. A major advantage over sequential loads is the fact that vector registers can be used for addressing the data; this means no detour of writing the contents of vector registers to the stack to move them into scalar registers required for sequential loads. In the following, we examine the behavior of the gather primitive in detail to evaluate its usefulness for the task of CT reconstruction.

At first glance (see Listing 3.5), the vgatherdps instructions looks similar to a normal load instruction, if it weren’t for the vector mask register k3. In the example zmm6 is the vector mask register.
3.3. Intel’s Many Integrated Core Microarchitecture

```c
#define INSTR vfmadd132ps
#define NINST 6
#define N edi
#define i r8d

.intel_syntax noprefix
.globl ninst
.data
ninst:
.long NINST
.text
.globl latency
.type latency, @function
.align 32
latency:
    push rbp
    mov rbp, rsp
    xor i, i
    test N, N
    jle done
loop:
    inc i
    INSTR zmm0, zmm5, zmm6
    INSTR zmm1, zmm0, zmm7
    INSTR zmm2, zmm1, zmm8
    cmp i, N
    INSTR zmm3, zmm2, zmm9
    INSTR zmm4, zmm3, zmm10
    INSTR zmm5, zmm4, zmm11
    jl loop
done:
    mov rsp, rbp
    pop rbp
    ret
.size latency, .-latency
```

Listing 3.4: Assembly code template to measure instruction latencies. By introducing artificial dependencies between successive operations in the loop (i.e. the first operation stores its result into zmm0, which is an input for the second operation; the second operation stores its result into zmm1, which is an input for the third operation; and so on) and across the loop (i.e. the last operation stores its result into zmm5, which is an input for the first operation) we make sure that no instruction can execute without waiting for the previous execution to finish, thereby measuring its latency. The inc, cmp, and jl instructions are executed on the scalar V-pipe and do not bias the measurement.
3. Intel 64 and Intel Many Integrated Core Architectures

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmovaps/pd</td>
<td>4</td>
</tr>
<tr>
<td>vaddps/pd</td>
<td>4</td>
</tr>
<tr>
<td>vmulps/pd</td>
<td>4</td>
</tr>
<tr>
<td>vfmadd{132,213,231}ps/d</td>
<td>4</td>
</tr>
<tr>
<td>vrcp23ps</td>
<td>6</td>
</tr>
<tr>
<td>vcvtfxpntps2dq</td>
<td>4</td>
</tr>
<tr>
<td>vrndfxpntps</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.3.: Latencies for various instructions used in the reconstruction kernel. Although we omit the fractional part of the measurement, we would like to note that obtained results were very precise (e.g., a value of 4.004 for the instructions with four cycles latency and a value of 6.004 for vrcp23ps using a frequency of 1047338 MHz as advertised by /proc/cpuinfo).

vgatherdps zmm6{k3}, [rdi + zmm13 * 4]

Listing 3.5: Single gather assembly instruction using Intel syntax.

registers in which data will be stored once it has been fetched. The rdi register contains a base address, zmm13 is a vector register holding 16 32-bit integers which serve as offset, and 4 is the scaling factor. The 16 bits of the vector mask register act as a write mask for the operation: If the nth bit is set to 1 the gather instruction will fetch the data pointed to by the nth component of the zmm13 register and write it into the nth component of the zmm6 register; if the bit is set to 0 no data will be fetched and the nth component of zmm6 is not modified. When the vgatherdps instruction is executed, only data from one CL is fetched. This means that when the data pointed to by the zmm13 register is distributed over multiple CLs vgatherdps has to be executed more than once. To check whether all data has been fetched, the vgatherdps instruction will zero out the bits in the vector mask register whenever the corresponding data was fetched. In combination with the jknzd and jkzd instructions—which perform conditional jumps depending on the contents of the vector mask register—it is possible to form loop constructs to execute the gather instruction as long as necessary to fetch all data, i.e. until the vector mask register contains all zero bits.

To benchmark the instruction, a set of kernels for likwid-bench were modeled on the instructions obtained by disassembling the _mm512_i32gather_ps intrinsic. A total of 5 kernels were designed: In the first, all data to be gathered resides in one CL, which means that the vgatherdps instruction has to be executed only once to fetch all data. In each successive kernel, the data is distributed over twice as many CLs. This means that in the second kernel, the 16 elements to be gathered are distributed over two CLs, and the vgatherdps instruction has to be execute twice. In the third, the data is distributed over four CLs, thus vgatherdps has to be executed four times, and so on...

Using all four hardware threads of a single core, these kernels were used to measure the
latencies to gather 16 elements from L1 cache, L2 cache, and main memory depending on how the data to be fetched is distributed. The results are shown in Table 3.4.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instruction</td>
<td>Loop</td>
<td>Instruction</td>
</tr>
<tr>
<td>16 per CL</td>
<td>9.0</td>
<td>90</td>
<td>13.6</td>
</tr>
<tr>
<td>8 per CL</td>
<td>4.2</td>
<td>8.4</td>
<td>9.4</td>
</tr>
<tr>
<td>4 per CL</td>
<td>3.7</td>
<td>14.8</td>
<td>9.1</td>
</tr>
<tr>
<td>2 per CL</td>
<td>2.9</td>
<td>23.2</td>
<td>8.6</td>
</tr>
<tr>
<td>1 per CL</td>
<td>2.3</td>
<td>36.8</td>
<td>8.1</td>
</tr>
</tbody>
</table>

Table 3.4.: Latencies in clock cycles encountered when gathering data from different levels of the memory hierarchy for different distributions of the data to be gathered. “Instruction” and “Loop” refer to the average latency of a single vgatherdps instruction and the total time required to gather all 16 elements.

We find that the latency for a single vgatherdps instruction varies depending on how many elements it has to fetch from a CL—at least when fetching data from the caches; the high memory latency seems to hide this effect when fetching the data from main memory. This might be taken as a hint that the vgatherdps is implemented as another loop in hardware—the higher the number of elements that have to be fetched in the loop, the higher the latency.

Nevertheless, we find that it is beneficiary for the data to reside in as few CLs as possible. Despite latency for a single vgatherdps instruction increasing with the number of elements per CL, the impact of having to call the vgatherdps instruction multiple times is far more severe—with the exception of the L1 cache, where using two vgatherdps instructions to gather data from two CLs is actually faster than one instruction to gather 16 elements from one CL.
Chapter 4

Fastrabbit on x86

This Chapter introduces the fastrabbit CPU implementation, which served as a starting point for the various Intel MIC implementations.

In the first Section, we describe the most important optimizations that were included in the original version published by Treibig et al. [14]. Among algorithmic improvements, these include vectorization and parallelization to make use of the full potential of the Sandy Bridge CPU architecture. While we try to elaborate on all aspects of the original implementation, our summary necessarily has to focus on those selected parts that will later be useful for our implementation on the Intel MIC; for a conclusive summary of the original fastrabbit we thus refer to the original paper [14].

After introducing the original optimizations, in Section 4.2 we describe the further improvements to the CPU version done as part of this thesis, which also found their way into the various implementations for the Intel MIC.

4.1. Existing Optimizations

4.1.1. Avoiding Redundant Calculations

One of the first optimizations is to move loop-invariant calculations out of the line update kernel. In our naive implementation, we calculated the image coordinates inside the innermost loop (cf. Listing 2.1, lines 4–11). However, the values of \( y \) and \( z \) and therefore the values of \( w_y \) and \( w_z \) do not change in the innermost loop, which iterates over \( x \).

This allows us to precalculate the loop-invariant parts of the image coordinates outside of the innermost loop (Listing 4.1, lines 4–10) and make use of these values once we entered the actual line update kernel (Listing 4.1, lines 15–17). This way, we can eliminate 12 floating-point operations (Flops) from the already compute-bound kernel.

\[ \text{In our FDK implementation we refer to the innermost loop, which updates one "line" of voxels of the volume in } x\text{-direction, as the "line update kernel."} \]
// iterate over all voxels in volume
for (z=0; z<L; ++z) {
    for (y=0; y<L; ++y) {
        // convert from VCS to WCS
        float wy = k + y * R;
        float wz = k + z * R;
        // convert from WCS to ICS
        for (x=0; x<L; ++x) {
            // convert from VCS to WCS
            float wx = k + x * R;
            // convert from WCS to ICS
            float ix = wx * A[0] + tmp_ix;
        } // x-loop
    } // y-loop
} // z-loop

Listing 4.1: FDK implementation avoiding redundant operations during the calculation of image coordinates in the innermost loop body. By precalculating the parts of the sum pertaining to the image coordinate calculation directly in the y-loop we can save 12 Flops in the line update kernel.
4.1.2. Zero Padding

As we have seen before (cf. Listing 2.1), the line update kernel uses if statements to make sure the projection data that is accessed lie inside of the projection image (lines 23–30); if the ray does not hit the detector, an intensity value of zero is assumed. The reason for this might not be clear at first glance. To illustrate the necessity for these conditional we refer to Figure 4.1.

Figure 4.1.: Not all rays from the x-ray source through the volume’s voxels are guaranteed to hit the detector. In this sample setup, we have highlighted one “line” of voxels, which would be processed by a invocation of the line update kernel. By copying the projection image into a zero-padded buffer of proper size, we can ignore the problem of outliers.

The CT device geometry does not guarantee that all rays through the volume’s voxels hit the detector. One way to deal with this shortcoming is the already mentioned use of conditionals; however, from a performance point of view, this is not an adequate solution, because the four if statements incur substantial instruction overhead.

A better solution to this problem is to copy the projection image into a zero-padded buffer that is large enough to “catch” all outliers, i.e. rays from the x-ray source through voxels that do not hit the flat-panel detector (cf. Listing 4.2). The necessary size of the buffer is easily determined by projecting the eight corners of the volume for all projections and check their
image coordinates before the actual reconstruction. Also, this padding can be determined once when calibrating the device setup, as the position of the virtual volume holding the object to be reconstructed will never change unless the device geometry is changed.

/* copy projection image into padded buffer before we process the projection */
float *offset = padding + pad_bottom * (pad_left + width + pad_right) + pad_left;
for (int i=0; i<height; ++i) {
    memcpy(offset + i*(pad_left+width+pad_right), I + i*width, width*sizeof(float));
}

Listing 4.2: Code to copy the original projection image contained in the buffer I into the zero-padded buffer padding.

We now no longer have to initialize the variables valbl, valbr, valtl, valtr holding the projection values and can also remove the boundary checks for the coordinates (see Listing 4.3). Whenever a ray projected through a voxel hits the detector, we load the correct intensity value from the copy of the original projection image in the padded buffer; in case the ray does not hit the detector, we are loading a value of zero from the zero-padded region of the buffer.

/* load four values for bilinear interpolation */
float valbl = I[iiy * pad_width + iix];
float valbr = I[iiy * pad_width + iix+1];
float valtl = I[(iiy+1) * pad_width + iix];
float valtr = I[(iiy+1) * pad_width + iix+1];

Listing 4.3: Code to load intensity values from the zero-padded buffer containing the original projection data. Note that we now use pad_width = pad_left + width + pad_right to access the data in the linearized array to compensate for the additional padding width introduced to the left and right of the projection data.

Although we are now required to copy the original projection data into the zero-padded buffer for each projection image, the benefit of removing the conditionals inside the line update kernels—which is executed $L^3$ times for each of the 496 projection images—far outweighs the cost of copying less than 5 MiB of projection data—which has to be performed only once per projection image.

4.1.3. Clipping Mask

As we have seen previously, whenever we project a ray through a voxel and we miss the detector an intensity value of zero is assumed and added to the voxel under consideration. Because adding zero to a voxel's intensity value essentially leaves the original value untouched and does not cause any change otherwise, we could simply skip over those voxels for a particular projection image by precalculating a clipping mask that contains a starting and stopping coordinate for each invocation of our line update kernel. Whenever the kernel is executed, instead of starting at voxel 0 and working to voxel $L - 1$, we start and stop at the precalculated
coordinates, thereby eliminating unnecessary work—which can be quite extensive. To get a better idea of the amount of work to be saved, Figure 4.2 shows all voxels that are projected onto the flat-panel detector in each of the 496 projection images. As can be seen, the number of voxels that have to be processed for all projections is only a fraction of the total number of voxels. Altogether, using the original clipping mask only 72.3% of all voxels have to be processed (for \( L = 512 \)).

![Figure 4.2: Frontal (left), isometric (middle), and top (right) view of voxels that contain information from all 496 projection images.](image)

The starting and stopping coordinates can each be stored in a variable of type `short` (2 byte), yielding a total of 4 bytes per row. Since the start and stop points depend on the current gantry position, we need to calculate the clipping mask for each projection image; also, we need to calculate the mask for all combinations of \( z \)- and \( y \)-coordinates, yielding a total memory footprint of \( 496 \cdot L \cdot L \cdot 4 \) bytes—e.g. 496 MiB for \( L = 512 \), the medically relevant case.

Unfortunately, due to vectorization, we still have to make use of the previously introduced zero-padding: Assume the precalculated starting position for a particular projection image and \( z \), \( y \)-coordinate to be 10. To avoid overhead like loop remainder handling, the line update kernel will always work on four (SSE) or eight (AVX) consecutive voxels at a time. To make that possible, the actual calculated starting and ending coordinates are set to be a multiple of four (SSE) or eight (AVX) in such a way that they include all the relevant voxels. In the case of SSE, the original value of 10 would be set to 8, because we can only skip processing voxels 0–3 and 4–7; the kernel iteration that will handle voxels 8–11 must be included, because voxel 10 is the first to lie inside the projection image and must be processed. In this case, voxels 8 and 9 mandate the use of the zero-padded buffer, because they lie outside the projection image.

### 4.1.4. Vectorization

In order to harness the full potential of a modern CPU architecture, vectorization is essential. Some parts of the algorithm are trivial to vectorize, while others require a lot of fine-tuning. We break up the algorithm into three different parts, which are separately discussed in more detail: The calculation of image coordinates (cf. lines 4–20 in Listing 2.1), the loading and successive interpolation of the intensity values (cf. lines 21–34 in Listing 2.1), and the update of the voxels in the volume (cf. lines 35–36 in Listing 2.1).
Calculate Image Coordinates

One of the easier parts is the calculation of image coordinates from where the four intensity values for the bilinear interpolation for each voxel are loaded. Instead of working on individual voxels, four (SSE) or eight (AVX) voxels are handled simultaneously.

Additionally, the divides in lines 13–14 of Listing 2.1 have been replaced by calculation of the reciprocal of $iw$ and successive multiplication with the reciprocal. The advantage of using this variant is that the reciprocal instruction ($vrcpps$) is pipelined with a throughput of one clock cycle in contrast to the original division; also, the reciprocal has a lower latency (7 clock cycles for $vrcpps$, compared to 17–21 clock cycles for $vdivps$ [18]), which may further prevent pipeline bubbles on 2-way SMT machines. The downside of substituting the division is that of decreased accuracy: The $vrcpps$ instruction does not calculate the exact reciprocal, but only an approximation with a maximum error of $1.5 \times 2^{-12}$. Despite the error, the quality of the resulting reconstruction is similar to that of GPU implementations.

Load and Interpolate Detector Values

When it comes to vectorization, this is the most difficult part of the algorithm, which is why we present a step-by-step guide through the implementation. In order to keep things simple, we focus on the SSE implementation—although the AVX version works almost identical.

As we can see in Figure 4.3, rays through contiguous voxels do not have to hit contiguous detector pixels.

We thus have to load the intensity values in a sequential fashion. Instead of loading only scalar values, we can make use of the fact that we always need two adjacent values for the bilinear interpolation (e.g. orange and blue, yellow and green values in Figure 4.3). We begin by loading the values as shown in Figure 4.4 using 64-bit loads ($movlps$, $movhps$).

After loading the intensity values, we have to prepare the interpolation weights. This involves rearranging the values inside the registers in such a way that they match the order of the intensity values (see Figure 4.3). In a first step, two vector registers ($xmm1$, $xmm2$) are derived by duplicating the low respectively high entries of the two 64-bit halves of the vector register containing the interpolation weights in $y$-direction ($xmm0$). In a second step, the vector registers containing the opposite weights are computed for $y$- ($xmm3$, $xmm4$), as well as $x$-direction ($xmm6$).

We then perform the interpolation in $y$-direction (see Fig. 4.6). After weighting the bottom ($xmm10$, $xmm11$) and top ($xmm12$, $xmm13$) intensity values with the appropriate interpolation weights ($xmm1$–$xmm4$), the results are added and we end up with the linearly interpolated values in registers $xmm10$ and $xmm13$.

Before we can perform the interpolation in $x$-direction, we have to reorder the data inside registers $xmm10$ and $xmm13$ (see Fig. 4.7). We start with register $xmm10$, which contains the left and right intensity values for voxel 0 in the lower half and the left and right intensity values for voxel 2 in the upper half. Again, we use the $movshdup$ instruction to duplicate the upper values inside the two 64-bit halves of the vector register. We end up with register $xmm14$, containing the right intensity value for voxel 0 in its lower half and the right intensity value for voxel 2 in its upper half. We then use the $blendps$ instruction and select alternating entries from the
Figure 4.3.: When projecting a ray from the x-ray source through contiguous voxels (0–3), the resulting intersections with the detector do not have to be contiguous pixels of the detector. For the interpolation we need the bottom left (valbl), bottom right (valbr), top left (valtl), and top right (valtr) intensity values.
4. Fastrabbit on x86

Figure 4.4.: Sequential loading of the intensity values required for the bilinear interpolations of four voxels. Here, the general purpose registers rax, rbx, r10d, and r11d contain the offsets to the bottom left values for voxels 0–3 respectively. Because we are using 64-bit loads, the bottom right value will be loaded as well. Registers rcx, rdx, r12d, and r13d hold the offsets to the top left values; again, we use 64-bit loads, thereby loading the top left and right intensity value simultaneously. We are left with four SSE registers: xmm10 contains the bottom intensity values for voxel 0 in the lower half of the register and the bottom intensity values for voxel 2 in the upper half; xmm11 contains the bottom values for voxel 1 and 3; xmm12 contains the top intensity values for voxel 0 in the lower half and the top intensity values for voxel 2 in the upper half; finally, xmm13 contains the top intensity values for voxels 1 and 3.

Figure 4.5.: Rearranging the interpolation weights in y-direction (into registers xmm1 and xmm2) to match the order of the intensity values inside the vector registers xmm10–xmm13. Afterwards, the opposite weighting factors are calculated by subtracting the newly arranged weights from a vector registers containing all floating-point ones (1.0f).
newly created register `xmm14` and register `xmm13`: We take the first value from register `xmm14`, which is the right intensity value for voxel zero; the second value is taken from `xmm13`, which is the right intensity value for voxel one; the third value is again taken from `xmm14`, which is the right intensity value for voxel 2; finally, the last value is taken from `xmm13`, which is the right intensity value for voxel 3. We thus end up with the right intensity values for voxels 0–3 in register `xmm14`. Similarly, we end up with the left intensity values for voxels 0–3 in register `xmm15`.

In the last step, we perform the interpolation in $x$-direction (see Fig. 4.8). We can now weight the correctly ordered intensity values in `xmm14` and `xmm15` with the appropriate interpolation weights (`xmm5` and `xmm6`) and add them. We are left with the results of the bilinear interpolation of all four voxels in `xmm14`.

**Update Volume Voxels**

All that is left now is to update the intensity values to the voxels. This is trivial now, because we have the intensity values in the correct order inside a vector register. We can simply use a vectorized add (`addps`) with a pointer to the memory holding the reconstructed volume as destination.
4. Fastrabbit on x86

Figure 4.7: Reordering of intensity values into sequential order. In the first step, we duplicate the high low respectively high entries of the two 64-bit halves; afterwards, we blend the registers to get the correct ordering.

Figure 4.8: Interpolation in $x$-direction: In the first step we correctly weight the left and right intensity values; in the second step we add the weighted values.
### 4.2. Additional Optimizations

#### 4.2.1. Clipping Mask

The original algorithm calculating the clipping mask had some minor flaws, which have been remedied. Using an improved version, the amount of clipping could be increased (see Table 4.1). For the medical relevant case $L = 512$ we can reduce the number of voxels that have to be processed by almost 10% when using the improved instead of the original clipping mask.

---

2The original algorithm that was used to calculate the clipping mask was based on a wrong assumption about the CT device geometry; in certain cases, this lead to voxel lines being included in the mask although none of the voxels were projected onto the detector.
Figure 4.9: Performance achieved using various scheduling strategies on the CPU reference hardware with Turbo Boost enabled and disabled. The red line shows the performance achieved in turbo mode with naive static scheduling and $L = 512$.

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>No Clipping</th>
<th>Original Clipping</th>
<th>Improved Clipping</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1,040,187,392</td>
<td>759,605,720 (73.0%)</td>
<td>695,562,032 (66.9%)</td>
</tr>
<tr>
<td>256</td>
<td>8,321,499,136</td>
<td>6,036,927,544 (72.5%)</td>
<td>5,488,873,928 (66.0%)</td>
</tr>
<tr>
<td>512</td>
<td>66,571,993,088</td>
<td>48,142,540,608 (72.3%)</td>
<td>43,625,981,928 (65.5%)</td>
</tr>
<tr>
<td>1024</td>
<td>532,575,944,704</td>
<td>384,507,594,848 (72.2%)</td>
<td>347,828,701,504 (65.3%)</td>
</tr>
</tbody>
</table>

Table 4.1: Numbers indicating the amount of work in comparison to an unclipped volume for both the original and the improved clipping mask. The “No Clipping” column refers to the total number of voxels ($496 \cdot L^3$) that have to be processed for a particular problem size when no clipping is used. The other clipping columns show the reduced number of voxels that have to be processed for a given implementation, as well as the fraction in relation to the unclipped version in parenthesis.
The improvement only involves clipping more unnecessary voxels from the volume, which means that the result of the reconstruction is identical to the original implementation.

4.2. Instruction Scheduling

The original assembly implementation of the line update kernel made use of almost all 16 available vector registers for the calculation of the voxel updates. Values required during the update like the image width (width), the partially calculated image coordinates (tmp_iχ, tmp_iy, tmp_iz), or parts of the projection matrix (\(A[0]\), \(A[1]\), \(A[2]\)) are passed over the stack to the line update kernel. In each loop iteration inside the line update kernel, these values were loaded again from the stack into vector registers, because the implementation overwrote these values during the computation of the voxel update.

By reusing registers which only contain temporary values as soon as these variables are no longer required, we can lower the amount of registers required for the actual computation. Thus, it becomes possible to buffer some of the arguments passed to the function across loop iterations in registers that are no longer needed for other purposes. This means we only have to load some of the arguments passed to the line update kernel once from the stack—before we enter the loop body—and then reuse the buffered values in all loop iterations instead of reloading them from the stack.

4.2.3. Parameter Handling

As described previously, in the original implementation parameters are passed over the stack to the line update kernel. In the calling function, this involved replicating scalar values onto the stack sequentially using scalar stores (see lines 21–33 in Listing 4.5).

A better way to perform parameter handling is to skip the stack altogether. The application binary interface (ABI) allows for passing values in up to eight vector registers to a function. Not only will this save the loads necessary to move the data from the stack into the vector registers inside the line update kernel, but also reduce the number of operations to get the data ready for the line update kernel: Instead of VECTORSIZE (e.g. 4 for SSE, 8 for AVX) scalar stores per variable—which need to access memory/cache—, we can use two (or even one in the case of SSE) operations to replicate the data directly inside registers—without going through the memory/cache subsystem.

The _mm256_set1_ps(float) instruction will duplicate a scalar float using two instructions: First, a shufps xmm0, xmm0, 0 will fill all four single-precision floating-point components of the 128 bit SSE part of an AVX register with the value found in the first component of the register, i.e. the scalar value; then, a vinsertf128 ymm0, ymm0, xmm0, 1 will copy the lower half of the AVX register (the legacy SSE mapping) into the upper 128 bit of the register.

Unfortunately, we can not pass all arguments inside vector registers, because the ABI only allows for a maximum of eight vector registers as function parameters but we have a total of nine arguments to pass as vectors to the line update kernel (\(wx, width, R, A[0], A[1], A[2], tmp_iχ, tmp_iy, tmp_iz\); cf. lines 24–32 in Listing 4.5). We chose the value width to be sequentially replicated onto the stack (like in the original version), because it is one of the values
// iterate over all voxels in volume
for (z=0; z<L; ++z) {
    for (y=0; y<L; ++y) {
        [...]   
    }   
}   

/* allocate aligned memory for contents of vector registers on the stack */
__declspec(align(VECTOR_BYTE)) float WX_VEC[VECTORSIZE];
__declspec(align(VECTOR_BYTE)) float MM_VEC[VECTORSIZE];
__declspec(align(VECTOR_BYTE)) float AO_VEC[VECTORSIZE];
__declspec(align(VECTOR_BYTE)) float A1_VEC[VECTORSIZE];
 [...]   
/* Contains three pointers.
* Each one points to VECTORSIZE elements
* of A[0], A[1], and A[2]. */
float *APTR[3];
APTR[0] = A0_VEC; // later filled with A[0], A[0], ...
APTR[1] = A1_VEC; // later filled with A[1], A[1], ...
APTR[2] = A2_VEC; // later filled with A[2], A[2], ...

/* Contains three pointers.
* Each one points to VECTORSIZE elements
* of tmp_ix, tmp_iy, and tmp_iz */
float *TMP[3];
TMP[0] = TMPIX_VEC; // later filled with tmp_ix, tmp_ix, ...
TMP[1] = TMPIY_VEC; // later filled with tmp_iy, tmp_iy, ...
TMP[2] = TMPIZ_VEC; // later filled with tmp_iz, tmp_iz, ...

/* prepare data for line update kernel */
#pragma unroll
for (int i = 0; i < VECTORSIZE; ++i) {
    WX_VEC[i] = wx + (float)i * R;
    WIDTH_VEC[i] = (float)width;
    R_VEC[i] = (float)VECTORSIZE * R;
    AO_VEC[i] = A[0];
    A1_VEC[i] = A[1];
    A2_VEC[i] = A[2];
    TMPIX_VEC[i] = tmp_ix;
    TMPIY_VEC[i] = tmp_iy;
    TMPIZ_VEC[i] = tmp_iz;
}
/* call line update kernel */
fastrabbit(...);
} // y-loop
} // z-loop

Listing 4.5: Code fragment showing the general idea of the original parameter passing to the line update kernel (fastrabbit).
that does not change across loop iterations and we therefore only have to set it once using \texttt{VECTORSIZE} scalar stores.

Also worth noting is the fact that despite now passing eight of the nine vector arguments inside vector registers, we still have to pass pointers to allocated memory on the stack because we need to do register spilling inside the line update kernel: Out of the total 16 vector registers, the optimized parameter handling is already blocking 8. To perform the voxel update, we have to back some of the parameters up onto the stack and later reload them in each loop iteration whenever necessary.

### 4.2.4. Results

Table 4.2 gives a summary of the improvements described in this Section. We have chosen to give performance numbers as time-to-solution rather than the previously-introduced GUP/s-metric, because this particular metric would have biased the enhancement of the improved clipping mask.

<table>
<thead>
<tr>
<th>Version</th>
<th>Best [s]</th>
<th>Average [s]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>9.75</td>
<td>9.77</td>
<td>—</td>
</tr>
<tr>
<td>Improved Clipping</td>
<td>8.85</td>
<td>8.87</td>
<td>1.10×</td>
</tr>
<tr>
<td>Instruction Scheduling</td>
<td>8.17</td>
<td>8.18</td>
<td>1.19×</td>
</tr>
<tr>
<td>Parameter Handling</td>
<td>7.86</td>
<td>7.88</td>
<td>1.24×</td>
</tr>
</tbody>
</table>

Table 4.2: Summary of improvements to the original version. The “Best” column shows the best performance (time-to-solution in seconds) achieved in 25 runs, while the “Average” column shows the average performance achieved during the same 25 runs. The cumulative speedup in relation to the original version is shown in the last column; the values are based on the average performance numbers. All runs performed with turbo mode enabled, \texttt{static,1} OpenMP scheduling, and thread-pinning via \texttt{likwid-pin} for $L = 512$.

We can see that the benefit in runtime of the refinement of the clipping mask directly relates to the amount of voxels that were additionally omitted: The number of voxels was reduced by a factor of $4.81 \cdot 10^{10} / 4.36 \cdot 10^{10} \approx 1.10$ (cf. Table 4.1); at the same time, the Time-to-Solution (TTS) was reduced by a factor of $9.77s / 8.87s \approx 1.10$ (cf. Table 4.2). The benefit of instruction scheduling was of the same quality as the clipping mask, while the parameter handling optimization had a smaller, but still noticeable, impact.

All things considered, the improved version of \texttt{fastrabbit} can perform the back projection about 1.25 times—corresponding to about two seconds—faster than the original version.

---

3 All the clipping mask improvement does is reduce the number of voxels that have to be processed by a certain amount. As we will see later in this Subsection, the runtime will decrease by that very same amount, leaving the original GUP/s value unchanged.
Chapter 5

Implementations for Intel MIC

In this Chapter, we present the different implementations for the Intel MIC architecture together with further architecture-specific optimizations that were not covered in Chapter 4. All the versions we present here rely on OpenMP for multi-core parallelization. Because efficient parallelization across cores is a more challenging task on the Intel MIC than it was on the Sandy Bridge CPU, we reserve the first Section of this chapter for a discussion on the subject.

In the remaining Sections, we explore different ways to perform vectorization, each requiring a different level of expertise to implement. We start out with scalar code that incorporates all but the vectorization optimizations from the last Chapter. We then guide the Intel compiler to vectorize this scalar code using two different approaches—compiler flags and the \texttt{#pragma simd} directive introduced in the latest OpenMP standard [16].

We continue with another auto-vectorization approach using the Intel SPMD compiler (ISPC) [24], which bears a striking resemblance to NVidia’s CUDA programming model [4]—although the authors of ISPC make every effort to point out the differences between the two.

Furthermore, we present a version written in C that was manually vectorized using intrinsics. Finally, we present an even more fine-tuned implementation of the line update kernel, written entirely in assembly, in the last Section.

5.1. OpenMP Parallelization

This Section revisits the subject of OpenMP parallelization in order to adapt some of the previous conclusions for the Intel MIC, as well as examine some aspects of parallelization—such as thread placement and the resulting data placement implications—that come into play when dealing with the Intel MIC.

In Section 4.1.5, we have seen that due to the imbalances caused by the clipping mask optimization it is necessary to lower the chunk size in order to allow for sensible load balancing between all threads. This is of course also true for the Intel MIC. Unfortunately, simply

---

1Single Program, Multiple Data—parallelism achieved by simultaneously executing multiple instances of the same program binary, each instance working on different data.
adapting the strategy from before (i.e. using a static scheduling with a chunk size of one) is not an option on the Intel MIC. For the clinically relevant case of $L = 512$, the number of planes is too small to be distributed to 240 threads, because a small number of stragglers would severely impact performance. In a perfectly balanced model, to process the 512 $x$-$y$-places each of the 240 hardware threads would work on approximately 2.13 planes. However, at a granularity of 1 slice per thread, we get 208 threads processing 2 planes and 32 threads processing 3 planes. Assuming a fixed amount of work per plane, this would cause a performance penalty of 29%.

This problem can be solved by making the amount of work more fine-granular—i.e. breaking down the planes into sub-planes or, in the most extreme case, single $x$-lines. Because the smaller units of work will take less time to complete, the penalty that stragglers—who have to process one additional unit of work—cause will be less significant in relation to the total runtime.

```c
#pragma omp parallel for schedule(runtime) collapse(2)
for (z=0; z<L; ++z) {
    for (y=0; y<L; ++y) {
        [...]
    }
}
```

Listing 5.1: Fusion of two loops using the #pragma collapse compiler directive.

A simple way to achieve this is the OpenMP #pragma collapse compiler directive (see Listing 5.1). The number in parenthesis tells the compiler how many of the directly following loops should be collapsed into a single loop. In our case, the $z$ and $y$ loop are merged into a single loop. Together with the #pragma omp parallel for directive, this means that a scheduling with chunk size of 1 will now distribute the volume line-wise to the threads, while a chunk size of 512 will distribute them plane-wise ($512$ contiguous lines form one plane for $L = 512$) as before.

The problem now of course is to find the right balance between minimizing the effect caused by stragglers and at the same time retaining a chunk size that is still dominated by useful computation, not by overhead.

Unfortunately, there exist a number of additional factors that have to be considered. For one, the chunk size influences the cache hit ratios. On Sandy Bridge, we did not have to bother about LLC hit ratios, because there the L3 cache was large enough to hold the entire projection image from which the detector intensity data is read during the line update kernel. As we have seen before, the Intel MIC does not have a shared LLC (cf. Section 3.3.2), so when the L2 caches (that may contain duplicate entries of the projection image) do not contain the data a thread is requesting a memory access is required. Depending on the chunk size, the amount of data that has to be fetched varies; larger chunks of the volume require a larger

---

2 At 1248×960 pixels, the projection images take up a only a small fraction (4.57 MiB) of the total L3 cache (20 MiB). Even with the zero-padding (13.15 MiB), the image still readily fits into the LLC.

66
5.1. OpenMP Parallelization

portion of the projection image to process the chunk, eventually leading to sizes that are too large to fit inside the cache.

The last factor we consider is that of thread placement to enable data sharing between the hardware threads. In Figure 5.1 we can see the difference between two thread placement strategies. The left side shows thread pinning and placement corresponding to likwid's logical numbering. In this logical numbering, likwid is using physical cores first to map OpenMP threads. The first 60 OpenMP threads will be mapped to hardware contexts 0 of cores 0–59; OpenMP threads 60–119 will be mapped to hardware contexts 1 of cores 0–59; OpenMP threads 120–179 will be mapped to hardware contexts 2 of cores 0–59; OpenMP threads 180–239 will be mapped to hardware contexts 3 of cores 0–59. This operation corresponds to a scatter setting of KMP_AFFINITY, which is why we will refer to this thread placement strategy as “scatter placement.” The problem with this approach is that the threads 0, 60, 120, and 180, which run on the four hardware contexts of core 0 have no spatial coherence in the voxel volume. As a result, they have no spatial coherence in the projection image as well. A direct implication of each hardware thread requiring different data from the projection image is that all four hardware threads will be competing for space in the L2 cache that is shared among them. The resulting preemptions and following reloads of preempted data from memory—which in turn preempts data of another thread—makes this placement strategy inefficient.

The right side of Figure 5.1 shows a better approach, which we call “compact placement”: Here, threads 0–3 are scheduled on core 0; threads 4–7 on core 1; …; and threads 236–239 on core 59. This corresponds to a compact setting for Intel’s KMP_AFFINITY setting. We can see that the portions of the projection image required by a group of threads running on the same physical core overlap. This avoids preemptions and results in a much more efficient cache usage—but only in the case the OpenMP chunk size is chosen carefully; if it is too large the amount of sharing diminishes.

In summary, the chunk size will influence a whole range of parameters that in turn induce

---

3 The environment variable KMP_AFFINITY is part of the Thread Affinity Interface that Intel recommends for thread pinning. 25
changes in the runtime: The impact of stragglers, the ratio of useful work to overhead, cache hit ratios, and the degree of cache sharing between threads. The combination and interaction of these factors make it difficult to chose the optimum chunk size analytically, and thus we determine it empirically.

![Graph of performance vs chunk size](image)

**Figure 5.2.** Different scheduling and thread placement strategies.

In Figure 5.2 we demonstrate the impact of different thread placement strategies on the performance of the line update kernel. The shapes of the graphs support our previous argumentation that the chunk size influences a number of factors that each contribute to the performance; the graphs are not straight lines or parabolas, but rather a superposition of different functions. The chunk size that resulted in the best performance was 262, which corresponds to approximately half a $x$-$y$-plane.

### 5.2. Intel C Compiler

As we have seen before (cf. Section 3.2.3), Intel compilers come with auto-vectorization support, which, however, is subject to several several restrictions—e.g. the ones imposed by the C language standard. To examine these restrictions, we begin by inspecting the scalar C code in Listing 5.2, which we later vectorize using the Intel C compiler.

The beginning of the loop body deals with the clipping mask optimization described in Section 4.1.3. In lines 5 and 6, we load the start- and end-points for the current voxel line in $x$ direction; if there is no work for the current $z$ and $y$ configuration, we skip over the loop iteration (line 7).
// iterate over all voxels
for (int z = 0; z < L; ++z) {
    for (int y = 0; y < L; ++y) {
        /* Select start and end coordinate for current line from the clipping mask */
        int start = Range[id - 1][(z * L + y).start;
        int stop = Range[id - 1][(z * L + y).stop;
        if (stop - start == 0) continue;

        /* Convert from VCS to WCS */
        float wz = z * MM + O;
        float wy = y * MM + O;
        float wx = start * MM + O;
        unsigned int offset = z*L*L + y*L;

        /* Precalculate parts of u, v, w that are invariant wrt to x */

        for (int x = start; x < stop; ++x, wx+=MM) {
            /* Calculate image coordinates */
            float u = tmp_u + wx*A[0];
            float v = tmp_v + wx*A[1];
            float w = tmp_w + wx*A[2];
            float rcp_w = 1.0f/w;
            float rcp2_w = rcp_w*rcp_w;
            float ix = u*rcp_w;
            float iy = v*rcp_w;

            /* Prepare interpolation weights */
            int iix = (int)ix;
            int iiy = (int)iy;
            float scalx = ix - (float)iix;
            float scaly = iy - (float)iix;

            /* Load intensity values from projection image */
            float valbl = I[iiy*pad_width + iix];
            float valbr = I[iiy*pad_width + iix+1];
            float valtl = I[(iix+1)*pad_width + iiy];
            float valtr = I[(iix+1)*pad_width + iiy+1];

            /* Perform bilinear interpolation and add result to voxel */
            float valb = (1-scalx) * valbl + scalx * valbr;
            float valt = (1-scalx) * valtl + scalx * valtr;
            float val = (1-scalx) * valb + scalx * valt;
            VOL[offset + x] += val * rcp2_w;
        } // x-loop
    } // y-loop
} // z-loop

Listing 5.2: FDK back projection kernel written in plain C.
Next, the voxel coordinates are converted to world coordinates in lines 9–12. In line 13, an offset that corresponds to the current position of the $z$-$y$-configuration inside the voxel volume is calculated; this saves unnecessary address calculations inside the innermost loop, when the calculated intensity values are updated inside the volume (line 46). Before we enter the line update kernel, the portion of the detector coordinates that is invariant of the $x$ coordinate is precalculated in lines 15–18 (cf. Avoiding Redundant Calculations optimization in Section 4.1.1).

The line update kernel (lines 20–47) works on voxels in accordance to the clipping mask, i.e. it begins with the voxel start and ends with the voxel stop. Inside the loop body, the first step is to complete the calculation of the detector coordinates (lines 21–24). The image coordinates are obtained by dehomogenizing the detector coordinates (lines 27–28). To perform this dehomogenization, we replace the division by a multiplication with the reciprocal (line 25); the weighting factor used later in the kernel (line 46) is also derived from this reciprocal (line 26). It is worth noting that the Intel MIC platform does not offer a “real” vectorized floating-point division—i.e. something similar to the vdivps and vdivpd instructions used by the Intel 64 architecture—in its instruction set. Instead, all division are replaced by multiplications with the reciprocal by the compiler.

After the computation of the image coordinates, the integer offsets into the projection images (lines 31 and 32) and the interpolation weights (lines 33 and 34) are calculated. The intensity values for the current voxel are then loaded from the current projection image in lines 36–40. Here, instead of using the original image width (width), we use pad_width—which corresponds to the sum of pad_left, width, and pad_right—as the width of the projection image when calculating offsets inside the buffer. This adjustment is made to account for the zero-padded buffer optimization to avoid conditionals inside the innermost loop (cf. Section 4.1.2).

Using the intensity values, the bilinear interpolation can be performed (lines 42–45); the result of the interpolation is then weighted and added to the existing value for the voxel in the volume (line 46).

The diagnostic information for this code output by the vectorizer can be obtained using the -vec-report3 compiler flag. We find that it is sufficient to specify the -mmic compiler flag—which instructs the Intel compiler to build a native MIC binary—for the code to be auto-vectorized. The reconstruction results of the vectorized code were verified against those produced by a scalar version, which can be generated by specifying the -no-vec flag. Instead of decreasing the PSNR and increasing the MSE, vectorization improved the results compared to the scalar version: The PSNR increased from 103.001 dB to 103.391 dB and the MSE decreased from 0.0289 to 0.0277.

Another approach is to use the new #pragma simd compiler directive from the latest OpenMP standard [16] to enforce the vectorization of the loop. While the original focus of OpenMP was to facilitate multi-core programming, the standard has evolved over time and the latest 4.0 version, which was recently released (July 23, 2013), includes concepts for SIMD.

---

4On a side note, there only exists a single-precision reciprocal operation: vrcp23ps; therefore, double-precision divisions have to be calculated using a lengthy combination of instructions (using several vrcp23ps instructions for a single result) that emulate the division.

5The part of the Intel compiler that deals with vectorization is known as the vectorizer.
vectorization, support for accelerators, and thread affinity management. Intel—as an active member in the OpenMP language committee—has already been including support for the \#pragma simd directive in its compilers for quite some time. This directive can be placed before any loop and will enforce its vectorization by overriding compiler heuristics that advise against vectorization; if a loop cannot be vectorized, the compiler will output a warning by default or exit with an error if the assert clause is used in the directive. In our case, we place the directive directly in front of the innermost loop (line 19 in Listing 5.2) to enable vectorization. Of particular interest is the fact that the loop vectorization is different from that of the native vectorizer. The performance achieved by the code generated using the \#pragma simd directive is slightly worse than that obtained with the native variant; also, the PSNR and MSE values are identical to those of the scalar implementation.

5.3. Intel SPMD Program Compiler

The ISPC is an open-source compiler developed by Intel—but separate from Intel’s main line of compiler products—for the standard C/C++ programming languages that features extensions for SPMD programming [24]. The extensions borrow from GPU programming models such as Compute Unified Device Architecture (CUDA) [4] and Open Computing Language (OpenCL) [5], in which the programmer writes so-called kernels, which appear to be serial programs. The execution model, however, is that multiple of the kernels will be run in parallel on the hardware. While CUDA and OpenCL both aim to provide parallelism at the core- and SIMD-level, ISPC restricts itself to SIMD vectorization. It currently supports the older Streaming SIMD Extensions Versions 2 and 4 (SSE2, SSE4), Advanced Vector Extensions Versions 1 and 2 introduced by Sandy Bridge (AVX1) respectively Haswell (AVX2), and the Initial Many Core Instructions (IMCI) developed for the Intel MIC.

In ISPC, kernels are written in a C-based programming language—which really is the standard C language, supplemented by a limited number of additional keywords—and compiled using the ispc binary. The resulting object file can then be used during linking to produce a binary. Upon entry to an ISPC function called from regular code, the execution model switches from the application’s serial model to ISPC’s SPMD model and the concurrently executing kernel instances are mapped on the SIMD vector units of CPUs, i.e. each kernel instance is running on a different SIMD lane.

Our full implementation of the FDK line update kernel for the Intel SPMD program compiler is shown in Listing 5.3.

The export qualifier in the first line of the listing is one of the additional keywords mentioned earlier; it indicates that this function will be exported so that it can be called by regular C/C++ code. Functions not marked with the export qualifier are intended to be called by other ISPC functions.

The uniform modifier before all variable names in the function signature (cf. lines 2–5 in Listing 5.3) indicates that these variables have the same values for all kernel instances; all values passed from regular C/C++ code must be uniform. Upon entry into the kernel, the scalar values that are passed into uniform variables are replicated using a mechanism similar to the one described before (cf. Section 4.2.3).
5. Implementations for Intel MIC

```c
export void Backprojection(uniform int start, uniform int stop, uniform float R, uniform int pad_width,
    uniform float A0, uniform float A1, uniform float A2, uniform float tmp_u,
    uniform float tmp_v, uniform float tmp_w, uniform float v,
    uniform float *uniform VOL, uniform float *uniform I)
{
    foreach (i = start ... stop) {
        // calculate world coordinate
        float wx = i*MM + O;

        // calculate image coordinates
        float u = A0 * wx + tmp_u;
        float v = A1 * wx + tmp_v;
        float w = A2 * wx + tmp_w;

        // guide ISPC to prefer reciprocal over division
        float rcp_w = 1.0f / w;
        float rcp2_w = rcp_w * rcp_w;

        // dehomogenize
        float ix = u * rcp_w;
        float iy = v * rcp_w;

        // calculate integral offsets, interpolation weights
        int iix = (int)ix;
        int iiy = (int)iy;
        float scalx = ix - (float)iix;
        float scaly = iy - (float)iiy;

        // load intensity values for interpolation
        float valtl = I[iiy*pad_width + iix];
        float valtr = I[iiy*pad_width + iix+1];
        float valbl = I[(iiy+1)*pad_width + iix];
        float valbr = I[(iiy+1)*pad_width + iix+1];

        // perform interpolation
        float valb = (1-scalx) * valbl + scalx * valbr;
        float valt = (1-scalx) * valtl + scalx * valtr;
        float val = (1-scaly) * valb + scaly * valt;

        // add weighted result to voxel
        VOL[i] += val * rcp2_w;
    }
}
```

Listing 5.3: FDK back projection kernel written for the Intel SPMD program compiler model.
5.4. C with Intrinsics

The last ISPC keyword in our kernel is the `foreach` loop construct (cf. line 7 in Listing 5.3), which guides the vectorization: On the Intel MIC, 16 loop instances will be processed in parallel, each one being mapped onto one of the 16 single-precision SIMD lanes of the 512-bit vector processing unit.

The rest of the kernel (lines 8–42) is plain C code and very similar to our line update kernel we presented in the last Section. We use the same clipping mask optimization as before: Instead of updating all voxels of a row we start processing the line at position `start` and end at `stop`. Inside the loop body, we begin by converting the `x`-coordinate of the into the world coordinate system in line 9, in order to finish the calculation of the image coordinates using the precalculated values passed to the function `\( (tmp_u, tmp_v, tmp_w) \)` in lines 11–14.

As we have established in the last Section, the Intel MIC instruction set does not offer division per se, but rather uses a combination of the `vrcp23ps` instruction—which computes the reciprocal—and a multiplication. Knowing this, explicitly pulling apart the division in the code (lines 17, 21–22) would not be necessary; however, to be platform independent we used this version. In line 18, the intensity weighting factor used later in the kernel is derived from the reciprocal.

In the next step (lines 20–22), the image coordinates are calculated by dehomogenizing the detector coordinates. Once the image coordinates have been computed, the integral offsets used for addressing inside the projection image are calculated (lines 25 and 26) and the interpolation weights are derived (lines 27 and 28). Next, the intensity values are loaded from the projection image; note that `pad_width` is again `pad_left+width+pad_right` (cf. Section 4.1.2), because we are using the zero-padded buffer optimization to avoid conditionals. After the bilinear interpolation (lines 36–39) the voxel's intensity value can be weighed and added to the existing value in the volume (lines 41–42).

5.4. C with Intrinsics

In contrast to the two previous auto-vectorization approaches, this implementation relies on intrinsic functions to perform manual vectorization.

Intrinsics are functions that are given a special treatment by the compiler. Just like inlined functions, they are replaced by the compiler with a set of instructions instead of actually calling a function; however, in contrast to inlined functions they are not substituted by instructions derived from user-written code—instead, intrinsic functions are replaced with a set of predefined, often highly optimized, instructions. While some intrinsics might provide a one-to-one mapping to allow the programmer to manually tap into special-purpose instruction sets such as vector extensions (e.g. Streaming SIMD Extensions (SSE), AVX) or cryptographic primitives (e.g. Advanced Encryption Standard – New Instructions (AES-NI), Intel Secure Hashing Algorithm (SHA) Extensions) others serve as entry points to optimized libraries (e.g. Short Vector Math Library (SVML)).

Because the implementation using intrinsics is quite lengthy, we chose to break the line update kernel down into three parts that we discuss separately: First, the calculation of the

---

6 In fact, we also ran the ISPC version on the Sandy Bridge architecture, where this notation is required to make the compiler chose the `vrcp2ps` and `vmulps` instructions over the standard divide (`vdivps`).

73
image coordinates and the interpolation weights; second, the gathering of intensity values from the projection images; and finally, the bilinear interpolation to compute the intensity value together with the update of the volume.

Before entering the line update kernel, all vector variables required inside the $x$-loop were properly replicated using the appropriate calls to _mm256_set1_ps. Unless otherwise noted, all variables inside the kernel that begin with an underscore are of type _m512, i.e. a 512-bit vector that holds 16 32-bit single-precision floating-point numbers. Again, the clipping mask optimization is used to reduce the number of loop iterations (start, stop in the line 4 of Listing 5.4); also note that the step size between loop iterations is set to 16 (cf. lines 1–4), which corresponds to the number of voxels that are processed simultaneously in one iteration.

```c
#define SIMD_BIT (512)
#define SIMD_BYTE (SIMD_BIT/8)
#define SIMD_OPS (SIMD_BYTE/sizeof(float))

for (int x = start; x < stop; x += SIMD_OPS) {
    // Calculate detector coordinates
    _u = _mm512_fmadd_ps(_wx, _A0, _tmp_u);
    _v = _mm512_fmadd_ps(_wx, _A1, _tmp_v);
    _w = _mm512_fmadd_ps(_wx, _A2, _tmp_w);

    // Calculate image coordinates and interpolation weights
    _rcp_w = _mm512_rcp23_ps(_w);
    _wx = _mm512_add_ps(_wx, _MM);
    _rcp2_w = _mm512_mul_ps(_rcp_w, _rcp_w);
    _ix = _mm512_mul_ps(_u, _rcp_w);
    _iy = _mm512_mul_ps(_v, _rcp_w);
    _iix = _mm512_round_ps(_ix, _MM_FROUND_TO_ZERO, _MM_EXPADJ_NONE);
    _iiy = _mm512_round_ps(_iy, _MM_FROUND_TO_ZERO, _MM_EXPADJ_NONE);
    _scalx = _mm512_sub_ps(_ix, _iix);
    _scaly = _mm512_sub_ps(_iy, _iiy);
    _1mscalx = _mm512_sub_ps(_one, _scalx);
    _1mscaly = _mm512_sub_ps(_one, _scaly);
```

Listing 5.4: First portion of the FDK line update kernel featuring intrinsics: Calculation of image coordinates and interpolation weights.

The first step inside the kernel is to finalize the computation of the detector coordinates (lines 6–8). For this task, we can make use of the fused multiply-add operation by applying the _mm512_fmadd_ps intrinsic. On the Intel MIC, this intrinsic maps to a single operation, which takes three arguments: It multiplies the first two arguments, then adds the product to the third argument.

After calculating the detector coordinates, we can derive the image coordinates by dehomogenization. To obtain the reciprocal of $w$, we use the _mm512_rcp23_ps intrinsics, which is mapped to the vrcp23ps instruction mentioned earlier.
Note that the calculation of the reciprocal (_rcp_w) is not directly followed by the computation of the weighting factor (_rcp2_w) as it was in the two previous implementations (plain C in Section 5.2 and ISPC in Section 5.3). While this may seem like a minor change, it is warranted and therefore on purpose: In the previous versions, the compiler took care of reordering instructions to avoid data hazards. We have argued before that this is rarely necessary, because most instructions have a latency of four clock cycles and using all four hardware threads of a core mitigates data hazards in this case (cf. Section 3.3.3). However, we have also measured that the vrcp23ps instruction is an exception to this rule and requires six clock cycles to complete (see Table 3.3 on page 46). As a consequence, the result of the vrcp23ps instruction is not ready for another two clock cycles by the time the hardware thread that issued the instruction regains control. To prevent this data hazard, we insert the _mm512_add_ps in line 12, which adjusts the world coordinates held in _wx for the next iteration. We can now dehomogenize the detector coordinates (_u and _v) using the _mm512_mul_ps intrinsic (lines 14–15) to obtain the image coordinates _ix and _iy.

To calculate the bilinear interpolation weights, which are given by the fractional parts of the image coordinates, we first have to make a copy of the original image coordinates and round them towards zero; afterwards, we subtract these rounded values from the originals to obtain the fractional parts. In our implementation, the rounding is done using the _mm512_round_ps intrinsic (lines 16 and 17). The first argument to this call is the vector of values to be rounded; the second argument governs the rounding rules, which in our case is _MM_FROUND_TO_ZERO to achieve rounding towards zero; the final argument can be used to perform exponent adjustments, which we do not require (_MM_EXPADJ_NONE). After rounding, all we have to do is to subtract the rounded values (_iix, _iiy) from their respective initial values (_ix, _iy) to obtain the interpolation weights. This subtraction is done using the _mm512_sub_ps intrinsic and the resulting weights are stored in the vectors _scalx and _scaly. In a final step, the opposite weights (_1mscalx, _1mscaly) are derived as well (lines 20–21).

After finishing the calculation of the image coordinates and the interpolation weights, we continue with the portion of the code that deals with the computation of the offsets inside the projection image and gathering the intensity values from these offsets.

The calculation of the offsets (cf. lines 22–26 in Listing 5.5) is based on the rounded image coordinates (_iix, _iiy). To obtain the offset to the bottom left intensity value, we multiply the y coordinate (_iiy) with the width of the projection image (_pad_width) and add the x coordinate (_iix); the realization of the multiplication and addition again take advantage of the fused multiply-add intrinsic (line 23). To derive the offset to the value to the right—i.e. the bottom right intensity value—we simply add a vector of all floating-point 1.0f values to the previously calculated vector (line 24). The offset to the top left value is obtained by adding the width of the image (_pad_width) to the offset to the bottom left value, thereby incrementing the y coordinate (line 25). As before, the addition of a vector containing all floating-point 1.0f values to the offset to the top left values yields the offset to the top right

7The values stored in the _MM vector are derived from the original scalar value inside MM and adjusted according to the loop stride of 16: _ms512 _MM = _mm512_set1_ps(16.0f * MM);
8Other possible values for this field include _MM_FROUND_TO_NEAREST_INT to round to the nearest integer, or _MM_FROUND_TO_NEG_INF and _MM_FROUND_TO_POS_INF to round towards negative and positive infinity respectively.
5. Implementations for Intel MIC

```c
// Calculate offsets to intensity values in the projection image
__bl_foffset = _mm512_fmadd_ps(_iiy, _pad_width, _iix);
__br_foffset = _mm512_add_ps(__bl_foffset, _one);
__tl_foffset = _mm512_add_ps(__bl_foffset, _pad_width);
__tr_foffset = _mm512_add_ps(__tl_foffset, _one);

// Convert floating-point offsets to integers that can be used as offsets for loading the data
__bl_offset = _mm512_cvtfxpnt_round_adjustps_epi32(__bl_foffset, _MM_FROUND_TO_ZERO, _MM_EXPADJ_NONE);
__br_offset = _mm512_cvtfxpnt_round_adjustps_epi32(__br_foffset, _MM_FROUND_TO_ZERO, _MM_EXPADJ_NONE);
__tl_offset = _mm512_cvtfxpnt_round_adjustps_epi32(__tl_foffset, _MM_FROUND_TO_ZERO, _MM_EXPADJ_NONE);
__tr_offset = _mm512_cvtfxpnt_round_adjustps_epi32(__tr_foffset, _MM_FROUND_TO_ZERO, _MM_EXPADJ_NONE);

// Use gather to load intensity values from the projection image
__valtl = _mm512_i32gather_ps(__tl_offset, I, 4);
__valtr = _mm512_i32gather_ps(__tr_offset, I, 4);
__valbl = _mm512_i32gather_ps(__bl_offset, I, 4);
__valbr = _mm512_i32gather_ps(__br_offset, I, 4);
```

Listing 5.5: Second portion of the FDK line update kernel: Determining offsets in the projection image and gathering data from there.
After all offsets have been calculated, they have to be converted into integer values in order to be used for addressing memory (lines 28–36). This conversion is done with the \_mm512\_cvtxfpnt\_round\_adjustps\_epi32 intrinsic. Its semantic is somewhat similar to that of the rounding operation: The first parameter contains the floating-point values that are to be converted. The second parameter allows to perform additional rounding before the conversion. In our case, we do not require any rounding, because all floating-point values used during the calculation of the offsets were integral values. Again, the third parameter can be used to perform exponent adjustments, which we do not require. The result of the conversion operation is a 512-bit vector integers of type \_m512i.

Instead of calculating the offsets with floating-point values and successively converting the offsets to integers another option would be to first convert coordinates into integers and calculate the offsets with them. The reason we chose the first approach was that we could reuse the floating-point vector \_one—which we already used to calculate the bilinear interpolation weights (cf. lines 20–21 in Listing 5.4)—to derive the top and bottom right offset from the top and bottom left offsets respectively. The second solution would require wasting an additional vector register, which has to be filled with all integer 1 values.

After the integer offsets have been computed, the intensity values can be loaded from the projection image (lines 38–42). The \_mm512\_i32gather\_ps uses its second parameter as a base pointer; the firsts argument contains the offsets, which are scaled by the third argument—in our case 4, because the floating-point values that are addressed are 4 bytes in size—before they are added to the base pointer. From these 16 resulting memory addresses, the floating point intensity values are fetched and placed inside a vector register.

After the intensity values for all 16 voxels have been loaded, we can proceed to the final part of the algorithm, which performs the bilinear interpolation, computes the voxels’ intensity values, and updates the 16 voxels in the volume.

```c
// Perform bilinear interpolation
_valtl = _mm512_mul_ps(_valtl, _1mscalx);
_valbl = _mm512_mul_ps(_valbl, _1mscalx);
_val = _mm512_fmadd_ps(_valtr, _scalx, _valtl);
_valb = _mm512_fmadd_ps(_valbr, _scalx, _valbl);
_valt = _mm512_mul_ps(_valt, _scaly);
_val = _mm512_fmadd_ps(_valb, _1mscaly, _valt);

// Weight interpolated intensity value and add to corresponding position in volume
_vol = _mm512_load_ps(&VOL[offset + x]);
_sum = _mm512_fmadd_ps(_val, _rcp2_w, _vol);
_mm512_store_ps(&VOL[offset + x], _sum);
}
```

Listing 5.6: Last portion of the FDK line update kernel: Calculate intensity values for the voxels and update the volume.

The bilinear interpolation, comprising of three linear interpolations, is performed first
5. Implementations for Intel MIC

(lines 43–49 in Listing 5.6). Each of the linear interpolations—requiring two multiplications and one addition—is implemented using one _mm512_mul_ps and one _mm512_fmadd_ps intrinsic.

Next, the current intensity values of the 16 voxels being processed is loaded from the voxel volume using the _mm512_load_ps intrinsic and stored in a vector register (line 52). A fused multiply-add intrinsic is then used to both weight the interpolated intensity value and add it to the existing value loaded from the volume (line 53). The final step of the algorithm is then to write the updated value back into the volume using the _mm512_store_ps intrinsic (line 54).

5.5. IMCI Assembly

In an attempt to achieve the best performance on the Intel MIC, our last implementation of the line update kernel is written in assembly. Using our knowledge of the architecture, this gives us the opportunity to perform fine-tuning on the lowest possible level.

Due to the complexity and length of the implementation, we split the kernel into multiple parts: We start with the function entry, parameter handling, and pre-loop initializations. We continue with the calculation of the image coordinates and the interpolation weights. Next in line is the calculation of offsets, used to address the intensity values in the projection image; this is then followed by loading the intensity values. Finally, the bilinear interpolation is performed and the data in the volume is updated.

In Listing 5.7 we can see how the kernel is called from inside the regular C code.

```
fastrabbit(I, VOL + offset, p_pad_width, p_float_one, N, _A0, _A1, _A2,
_tmp_u, _tmp_v, _tmp_w, _wx, _MM);
```

Listing 5.7: Invocation of the FDK kernel from C code.

The arguments passed to the kernel include:

- I, a pointer to the current projection image;
- VOL+offset, a pointer to voxel volume that has already been adjusted to point to the current x line of voxels that are being updated;
- p_pad_width, a pointer to a 64-byte aligned memory address at which the value pad_width is stored successively 16 times;
- p_float_one, another pointer to a 64-byte aligned address at which the floating-point value 1.0f is stored successively 16 times;
- N, which corresponds to the loop limit (i.e. the difference of the variables containing the clipping information: stop - start);
- _A0–_A2, 16-vectors (i.e. variables of type __m512) containing the value of A[0]–A[2] respectively in each component;
- _tmp_u, _tmp_v, and _tmp_w, another set of 16-vectors containing the values tmp_u, tmp_v, and tmp_w respectively in each component;
5.5. IMCI Assembly

- \_wx, a 16-vector which has been set to contain the world coordinates of the first 16 voxels to be processed (already compensating for the clipping mask improvement) as shown in Listing 5.8 and
- \_MM, a 16-vector which contains the appropriate values to adjust the contents of \_wx to the next 16 voxels for the next loop iteration. In the scalar version, we were incrementing a voxel’s world coordinates by adding the voxel width (\_MM) to it: \texttt{wx+=\_MM}. Now that we are processing 16 voxels in one loop iterations, the correct increment is 16 \cdot \_MM; the 16-vector can thus be set using \texttt{\_m512 \_MM = \_mm512_set1_ps(16.0f * \_MM);}.

Listing 5.8: Setting the \_wx vector for the assembly kernel: The starting position \texttt{start} is extracted in line 4 and used to calculate the world coordinates \texttt{wx} in line 6; the content of this scalar variable is then replicated into the 16-vector \_wx in line 7. After the \_setup vector (line 1) has been added to \_wx in line 8 the vector contains the world coordinates of the first 16 voxels to process.

Unfortunately, the Intel MIC Application Binary Interface (ABI) only allows to pass up to eight vector registers as function parameters, which is why we have to make a slight detour in order to pass some of the data required in the line update kernel. The data in question is the vector containing the padded image width (\texttt{pad\_width}), which is required during the offset calculation, and the vector containing all floating-point \texttt{1.0f} values, which we need to calculate the opposing interpolation weights and again during offset computation. The detour consists in allocating two times 64 bytes of 64-byte aligned memory that are sequentially filled with the appropriate values before the line update kernel is called; then, inside the line update kernel the vector registers can be filled with the data stored at these addresses. The reason the width and the \texttt{1.0f} vector were chosen is that they are invariant during the outer loops (i.e. the \texttt{z} and \texttt{y} loop); therefore, it is sufficient to initialize them only once, which avoids unnecessary overhead in the outer loops. This is similar to the optimization we performed on the original CPU version of \texttt{fastrabbit} (cf. Section 4.2.3).

To recapitulate, Table 5.1 contains a mapping of both scalar and vector parameters corresponding to the function call show in Listing 5.7.

After gaining a thorough understanding of the parameter handling, we can begin our examination of the pre-loop initializations performed in the assembly code.

As we can see in Listing 5.9, the amount of work to set up the function is minimal. Lines 1–5 contain some information meant for the compiler (the first line for example indicating
Register | Content
--- | ---
rdi | Pointer to the projection image \( (I) \).
rsi | Pointer to the current line of voxels to be updated in the volume \( (\text{VOL}+\text{offset}) \).
rdx | Pointer to a 64-byte aligned memory address at which the value \( \text{pad\_width} \) is stored 16 times.
rcx | Pointer to a 64-byte aligned memory address at which the value \( 1.0f \) is stored 16 times.
r8d | The loop limit \( (N) \).

\[
zmm0 \quad \text{All SIMD-lanes of the vector register are filled with the value stored in A[0].}
zmm1 \quad \text{All SIMD-lanes of the vector register are filled with the value stored in A[1].}
zmm2 \quad \text{All SIMD-lanes of the vector register are filled with the value stored in A[2].}
zmm3 \quad \text{All SIMD-lanes of the vector register are filled with the value stored in tmp_u.}
zmm4 \quad \text{All SIMD-lanes of the vector register are filled with the value stored in tmp_v.}
zmm5 \quad \text{All SIMD-lanes of the vector register are filled with the value stored in tmp_w.}
zmm6 \quad \text{World coordinates for the first 16 voxels to be processed (wx).}
zmm7 \quad \text{All SIMD-lanes of the vector register are filled with the value stored in MM, multiplied by 16.}
\]

Table 5.1: Registers holding the arguments passed to the FDK kernel.

```c
.intel_syntax noprefix
.text
globl fastrabbit
.type fastrabbit, @function
.align 32
fastrabbit:
    vmovaps     pad_width, [p_pad_width]
    vmovaps     float_one, [p_float_one]
    xor         i, i
    vprefetchenta [VOL]
    vprefetch2  [VOL + 256]
```

Listing 5.9: Compiler information and pre-loop initialization of the FDK kernel.
that we use Intel syntax). Line 6 marks the function entry point. During the design of the function, we made sure not to use any callee-save\(^9\) registers in our code, which is why we can skip pushing callee-save registers onto the stack.

Inside the function, instead of using register names for the vector registers (e.g. \(zmm0\)–\(zmm31\), \(rdi\), \(rsi\), etc.) we chose to use symbolic names in order to improve the readability of the code. This is implemented using a set of preprocessor defines (e.g. \#define AO zmm0) to map the symbolic name \(AO\) to vector register \(zmm0\). A table showing the mapping of symbolic names to registers can be found in Appendix \(A\).

Before we can enter the \(x\) loop, we have to prepare the two vector registers \(pad\_width\) and \(float\_one\) by filling them with the data that were precalculated prior to calling the function. By using aligned memory before, we can now make use of the aligned move instruction \(vmovaps\) (lines 7 and 8); another requirement is to set the loop counter \(i\) to zero, which is done with the \(xor\) instruction (line 9). The \(vprefetch\_n\) instruction (line 10) prefetches the part of memory that is required at the end of the line update kernel when updating the voxel volume. This particular prefetch brings the data into the non-temporal portion of the L1 cache with intention of writing to it using the exclusive hint (cf. Table \(3.2\) on page \(38\)). The data is fetched into the non-temporal cache, because we do not want to pollute the normal L1 cache with it, because we know for certain that the data would have been evicted from the cache by the next time we need it. Taking into account the clipping mask, which spares us from processing of 34.5\% of the volume (cf. Table \(4.1\) on page \(60\)), we still need to process about 335 MiB of volume data per projection image for the clinically relevant case of \(L = 512\) (\(512^3\) Voxels \(\cdot\) 4 Byte/Voxel \(\cdot\) 65.5 \%)—much too large for any of our caches. The exclusive hint is used, because we always write to the address when updating the voxel intensity values in the volume. Contention is not a problem because we decompose the volume geometrically for the parallelization; each core is processing its own private sub-volume. In a similar fashion, the \(vprefetch\_n\) instruction prefetches the volume data that will be required after four loop iterations in the non-temporal part of the L2 cache with the exclusive hint.

Next in line is the calculation of the image coordinates and interpolation weights shown in Listing \(510\).

To compute the image coordinates, we begin by making three copies of the contents of the \(wx\) register (lines 13, 15, and 17). These copies are required, because the \(vfmadd213\_ps\) used later is destructive, i.e. it overwrites the contents of one of its arguments with the result of its computation. However, we still need the contents of \(wx\) for the next loop iteration, which is why we make the copies. Note that each of the \(vmovaps\) instruction is directly followed by an unrelated \(kxnor\) instruction (lines 14, 16, and 18). Both the \(vmovaps\) and \(kxnor\) instructions belong

---

\(^9\)In the Intel MIC\(^{AB}\)—as well as most other ABIs—there is a differentiation between caller-save and callee-save registers. To preserve values inside registers across function calls, there is a convention that requires to store the contents of some registers onto the stack before calling a function; these variables are then restored after the function has finished, thus guaranteeing their value after the function has completed is consistent with the value from before the function call. These are known as caller-save registers. Complementary, there exist a number of callee-save registers, which are not saved on the stack before a function call. The function being called is responsible for saving the contents of these registers onto the stack before they are being used; the called function also has to restore the contents of the registers (by popping them from the stack) before returning control to the calling function.
5. Implementations for Intel MIC

..loop:

```
vmovaps w, wx
kxnor k2, k2
vmovaps v, wx
kxnor k3, k3
vmovaps u, wx
kxnor k4, k4
vfmadd213ps w, A2, tmp_w
kxnor k5, k5
vrcp23ps w_inv, w
lea r11, [r10]
vfmadd213ps u, A0, tmp_u
add i, 16
vfmadd213ps v, A1, tmp_v
vmulps ix, u, w_inv
vmulps iy, v, w_inv
vrndfxfpntps iiy, iy, 3
vrndfxpntps iix, ix, 3
vmulps w2_inv, w_inv, w_inv
vsubps scalv, iy, iiy
vsubps scalx, ix, iix
```

Listing 5.10: Calculation of the image coordinates and interpolation weights.

of the set of vector instructions that can be executed on both the vector U-pipe (connected to the VPU), as well as the scalar V-pipe (cf. Table 3.1 in Intel Xeon Phi Coprocessor Vector Microarchitecture [27]). By grouping these instructions, they are paired in the CPU frontend and execute simultaneously on both pipelines. The kxnor instructions prepare the vector mask registers that are required later in the kernel for the gather operation by setting all bits to one by using the inverse of the exclusive or function (xnor); the k prefix identifies the instruction as a vector mask operation.

After the copies of the wx register have been created in the u, v, and w registers, we can finalize the calculation of the detector coordinates. This is done with the three vfmadd213ps instructions in lines 19, 23, and 25. We start with the calculation of w, because we have to calculate the reciprocal of it as soon as possible to minimize the critical path of our kernel. The instruction is again paired with one of the kxnor instructions (line 20); this time, the kxnor has to execute on the scalar V-pipe, because the vfmadd213ps must be dispatched on the VPU, which is connected to the U-pipe.

The next instruction (vrcp23ps in line 21) calculates the inverse of w. It is paired with the scalar lea (load effective address) instruction which duplicates the address stored in r10 into the r11 register. The reason we chose to use register names instead of symbolic names for this instruction is to highlight the intricacies that are involved. Our loop is implemented using a counter—in our case i—which gets incremented (add i, 16 in line 24); it is then compared to the loop limit (cmp i, N in line 64 in Listing 5.14) to determine whether to jump and
execute the loop again or leave the loop. On a Sandy Bridge processor, all these instructions can happen at the end of the loop, right before jumping back, and no performance penalty is experienced, because usually the out-of-order window will execute the loop increment instruction long before the jump has to be performed and the macro-op fusion will combine the compare and branch instruction into a single micro-op. On the Intel MIC, it is more complicated to get the loop handling overhead done without performance penalty, because we have an in-order design and no macro-op fusion. If we would just put the increment, compare, and branch instructions after the actual work performed by the loop, we would prolong the loop by three clock cycles, because each of the instructions has to be scheduled on the V-pipe and no macro-op fusion occurs. One way to prevent performance penalties is to hide the scalar instructions by pairing them with vector instructions. What makes the loop management even more complicated in our case is the fact that we store the 32-bit integer that serves as the loop counter in the lower 32-bit of the 64-bit general purpose register that we simultaneously use to address memory inside the voxel volume. Unfortunately, the last vector instruction in the kernel needs to perform such addressing when writing the result of the kernel back into the voxel volume; therefore, we can not increment the loop counter before this last vector instruction. The solution to this problem is the duplication of the loop counter using the `lea` instruction. This way, we can use the `r11` register, which contains a copy of the loop counter before it is increased, to do the addressing in the vector instruction at the end of the kernel. Because the loop counter in `r10` is now no longer used in addressing, it is thus possible to increment it in due time (line 24, paired with the vector instruction in line 23).

We can now compute the image coordinates by dehomogenizing the detector coordinates (lines 26 and 27). To obtain the interpolation weights, we need to round the floating-point image coordinates towards zero. This is accomplished using the `vrndfzpntps` instruction in lines 28 and 29; the third argument to the instruction is a constant that determines the rounding mode—in our case rounding toward zero. We insert the computation of the weighting factor for the voxel's intensity value in line 30 before we continue to compute two of the interpolation weights in lines 31 and 32.

```
vfmadd213ps iiy, pad_width, iix
vaddps off_tl, pad_width, off_bl
vaddps off_br, float_one, off_bl
vaddps off_tr, float_one, off_tl
vcvtfxpntps2dq ioff_bl, off_bl, 3
vcvtfxpntps2dq ioff_br, off_br, 3
vcvtfxpntps2dq ioff_tl, off_tl, 3
vcvtfxpntps2dq ioff_tr, off_tr, 3
```

Listing 5.11: Computation of the offsets in the projection image.

Using the image coordinates, we can now compute the offsets into the projection images. In Listing 5.11 we can see that this is a two step process involving the computation of the offsets

10Using the loop counter for addressing saves us from updating two registers per loop iteration.
5. Implementations for Intel MIC

followed by a conversion from floating-point to integers. We begin with a fused multiply-add operation in line 33. The result is stored in the iiy registers. Because the iiy and off_tl symbols both reference the same vector register (cf. Appendix A), we can refer to the result of the operation using the off_tl symbol. This is what we do in the next instruction (line 34): To obtain the offsets to the top left values (off_tl) we add the projection image width to the offsets addressing the bottom left values (off_bl). To obtain the right values we just add a value of one to the left offsets (lines 35 and 36). In the next step, the offsets—currently in floating-point format—are converted to 32-bit integers using the vcvtfxpntps2dq instruction; again, the third parameter is a constant that tells the instruction to perform towards-zero-rounding when performing the conversion.

Listing 5.12: Gathering intensity values from the projection image.

Using these integer offsets, we can gather the intensity values from the projection image. Listing 5.12 shows the gather loop construct for the top bottom left (lines 41–44), bottom right (lines 46–49), top left (lines 51–54), and top right (lines 56–59) intensity values. To better understand the gather loop construct, we will discuss the bottom left instance in lines 41–44 in more detail.

The vgatherdps instruction uses one vector mask registers set earlier (cf. line 14 in Listing 5.10). It uses the 16 bits of the mask register to determine which off the 16 elements in the val_bl register should be filled with the data pointed at by its second argument I+ioff_bl*4. Because we initialized the vector mask register using the kxnor instruction, all 16 components of the vector register will be filled. Instead of fetching all data with one instruction, vgatherdps works by getting the data CL-wise per invocation; this means that every time the
gather instruction is used, it will fetch only one CL, load all the values that it is supposed to gather from this CL, store them in the destination vector register, and finally zero out the bits of the components that have been filled in the vector mask register. As a consequence, the number of gather instruction depends on the distribution of the data pointed at by ioff_bl. If all data resides in one CL then one gather instruction is enough; in the worst case, each intensity value is located in a different CL which will require sixteen gather instructions. The jkzd instruction in line 42 checks the contents of the vector mask register that was updated in the line before by the vgatherdps instruction. If the mask register is zero—i.e. all zero bits indicating that all data has been gathered—control flow continues at label ..L101 in line 46; if the mask register is non-zero—i.e. one or more one bits set, indicating that there is still data to be fetched—no jump is performed and the vgatherdps in line 43 is scheduled next. After more data has been gathered and k2 has been updated, the vector mask register is tested again for zero by the jknzd in line 44. If there is still data to be fetched, control flow will jump back to label ..L100 in line 41, starting the gather construct all over; if all data has been fetched, the construct is left.

```
1  ..L100: vgatherdps  val_bl{k2}, [I + ioff_bl * 4]
2    jknzd           k2, ..L100
```

Listing 5.13: Alternative loop gather construct.

In Listing 5.13, we can see that a single gather, followed by a check whether to jump back and perform the gather again—which is a much simpler and more obvious approach to the problem—gets the same work done as the construct described before. We found, however, that this simpler handler resulted in a much worse performance. The more complicated construct was obtained by analyzing the assembly code generated by the compiler for the _mm512_i32gather_ps intrinsic. Because its performance is much better, we chose to include it in our handwritten assembly code instead of the variant shown in Listing 5.13. Unfortunately, we could not get a satisfactory answer from Intel as to why this performance penalty occurs.

Between the gather constructs for the bottom left, bottom right, top left, and top right intensity values various instructions have been inserted (e.g. the computation of the opposite interpolation weights in lines 45 and 50 or the increment of the world coordinates in line 55).

After all intensity values have been gathered, the bilinear interpolation begins (cf. lines 61–67 in Listing 5.14).

The bottom and top left intensity values are scaled using the appropriate interpolation weights in lines 61 and 63. The first multiplication is paired with the scalar cmp instruction; the result of this comparison is used by the jl (jump if less) instruction in line 73 that jumps back to the ..loop label in line 12 of the kernel if \( i \) is smaller than \( N \). The vfmadd213ps instructions in lines 64 and 65 combine the scaling of the bottom and top right intensity values with the appropriate weight and the addition to the already weighted bottom and top left intensity values (sval_bl, sval_tl); this completes the first two linear interpolations in \( x \)-direction and
We can now perform the bilinear interpolation in $y$-direction by first scaling the bottom intensity value in line 67; next, we again combine the scaling of top intensity value with the addition to the already scaled bottom intensity value using the fused multiply-add instruction in line 67. We are thus left with the final interpolated value in the register $val$, which references the same vector register as the $val_t$ symbol.

After computation of the bilinear interpolation all that is left is to weight the value and update the voxel volume. Part of this is done using the $vfmadd213ps$ instruction in line 68; note that the register used in addressing here is the $r11$ register, which holds a copy of the current loop counter. In the next line, the data at the memory address that is required in the next iteration is prefetched into the non-temporal portion of the L1 cache using the exclusive hint; this is done by using the $r10$ register, which holds the already incremented loop counter. Similarly, the $vprefetch2$ instruction prefetches the date required in four loop iterations into the non-temporal part of the L2 cache using the exclusive hint. Finally, we have to write the updated intensity values of the 16 voxels back into the volume. This is done in line 72, again using the $r10$ register. This final vector store is paired with the scalar jump in line 73; to achieve this pairing a $nop$ operation has been inserted in line 71. Omitting the $nop$ would cause the $vmovaps$ instruction to pair with the $vprefetch2$ from line 70. This would lead to a situation in which two instruction try to access the cache at the same time, which is impossible. The result is that only one operations can access the cache, the other will be stalled. If the instruction being stalled is scheduled on the scalar V-pipe, the $jl$ from line 73 will have to wait an additional clock cycle for the V-pipe to become available.

Line 74 contains the function epilogue, while line 75 contains information for the compiler.

---

Listing 5.14: Bilinear interpolation and update of the voxel volume.

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>$vmulps$</td>
<td>sval_b, val_b, oscalx</td>
</tr>
<tr>
<td>62</td>
<td>$cmp$</td>
<td>i, N</td>
</tr>
<tr>
<td>63</td>
<td>$vmulps$</td>
<td>sval_t, val_t, oscalx</td>
</tr>
<tr>
<td>64</td>
<td>$vfmadd213ps$</td>
<td>val_br, scalx, sval_bl</td>
</tr>
<tr>
<td>65</td>
<td>$vfmadd213ps$</td>
<td>val_tr, scalx, sval_tl</td>
</tr>
<tr>
<td>66</td>
<td>$vmulps$</td>
<td>sval_b, val_b, oscaly</td>
</tr>
<tr>
<td>67</td>
<td>$vfmadd213ps$</td>
<td>val_t, scalx, sval_b</td>
</tr>
<tr>
<td>68</td>
<td>$vfmadd213ps$</td>
<td>val, $w2_{inv}$, [VOL + $r11 \times 4$]</td>
</tr>
<tr>
<td>69</td>
<td>$vprefetchenta$</td>
<td>[VOL + $r10 \times 4$]</td>
</tr>
<tr>
<td>70</td>
<td>$vprefetch2$</td>
<td>[VOL + $r10 \times 4 + 256$]</td>
</tr>
<tr>
<td>71</td>
<td>$nop$</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>$vmovaps$</td>
<td>[VOL + $r11 \times 4$], val</td>
</tr>
<tr>
<td>73</td>
<td>$jl$</td>
<td>..loop</td>
</tr>
<tr>
<td>74</td>
<td>$ret$</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>.size fastrabbit, .-fastrabbit</td>
<td></td>
</tr>
</tbody>
</table>

Note that the symbolic name $val_b$ references the same vector register as the $val_br$ symbol used in line 64; similarly, the symbolic name $val_t$ references the same vector register as the $val_tr$ symbol in line 65.
6.1. Performance Model

Popular performance models like the Roofline model \[28\] reduce investigations to determining whether kernels are compute- or memory-bound, not taking runtime contributions of the cache subsystem into account. The performance model we use is based on a slightly modified version of the model used by Treibig et al. in the original fastrabbit publication \[14, 29\]. At the basis of the model is the execution time required to update the 16 voxels in a single CL, assuming all data is available in the L1 cache. In addition, the contribution of the cache and memory subsystem is modeled, which accounts for time spent transferring all data required for the update into the L1 cache and back. In the original model, designed for out-of-order CPUs, an estimation whether the cache subsystem overhead can be hidden by overlapping it with the execution time is given and the authors conclude that there exist sufficient suitable instructions\(^1\) to hide any overhead caused by in-cache transfers. However, in their analysis, Treibig et al. only consider the in-cache contribution of the CLs relating to the voxel volume; all CLs pertaining to the projection images, required for the bilinear interpolation, are assumed to reside in the L1 cache. On the Intel MIC we find this simplification no longer holds true. There is a non-negligible cost for transferring the projection data from the L2 to the L1 cache that can not be overlapped with the execution time.

6.1.1. Core Execution Time

For the instruction analysis of our model, we neglect the facts that one core contains four hardware contexts and that no hardware context can issue instructions in two successive clock cycles. Using this abstracted view, the core, which can execute instructions any given clock cycle, becomes the resource of interest for our investigation.

Unfortunately there exist no tools such as, e.g., the Intel Architecture Code Analyzer (IACA) \[30\], which is used to measure kernel execution times on Intel's CPU microarchitectures, for  

\(^1\)Because the L1 cache is single-ported—i.e. it can only communicate with either the core or the L2 cache at any given clock cycle—transfers between the L1 and L2 caches can only overlap with “suitable” instructions that do not access the L1 cache such as, e.g., arithmetic instructions with register operands.
the Intel MIC. Therefore, we have to perform a manual estimation of the clock cycles spent on the line update. To complicate things, simply counting instructions in the line update kernel is not an option, because the number of vgatherdps instructions varies depending on the distribution of the data to be fetched across CLs. As a consequence, we begin with an estimation of the execution time for a gather-less kernel (i.e. a version of the line update kernel in which all gather loop constructs have been commented out).

Bearing in mind instruction pairing, the first part of the kernel, which deals with calculating the image coordinates and interpolation weights (cf. Listing 5.10 on page 82), contains a total of 14 instructions for the U-pipe. The computation of the offsets in the projection image contains 8 instructions (cf. Listing 5.11). Another 3 instructions are required to calculate the opposite interpolation weights and increment the world coordinates (cf. lines 45, 50, and 55 in Listing 5.12). Again bearing instruction pairing in mind, the final part of the algorithm, which performs the bilinear interpolation and updates the voxel volume, contains 9 instructions for the U-pipe (cf. Listing 5.14). In total, one iteration of the gather-less line update kernel, corresponding to a CL update, requires 34 clock cycles.

The analytical estimation of 34 clock cycles was verified by measurement. For one voxel line containing 512 voxels a runtime of 2402 clock cycles was obtained using a single thread. This corresponds to 75 clock cycles per iteration. Taking into account that the single thread can only issue instruction every other clock cycle, the core execution time for one loop iteration is approximately 37.5 clock cycles—which is a close fit to the value of 34 clock cycles determined previously. For our model, we use the measured value of 37.5 clock cycles because it contains non-negligible overhead that was not accounted for in the analytical value.

To estimate the contribution of the gather loop constructs we first determine how often a vgatherdps instruction is executed on average for a CL update. To get this value, we divide the total number of gather instructions issued during the reconstruction by total number of loop iterations.

The measured number of gathers, \(4.36 \times 10^{10}\), offers some insight into the average distribution of the intensity data for the 16 consecutive voxels that are handled in one loop iteration: For the reconstruction, we have to process \(4.39 \times 10^{10}\) voxels. At a processing rate of 16 voxels per loop iteration this results in a total of \(2.75 \times 10^9\) iterations to perform the reconstruction. Splitting the \(4.36 \times 10^{10}\) gather invocations among the kernel iterations yields an average of 16 vgatherdps instructions per iteration. Further distributing the 16 gather instructions among the 4 loop gather constructs in each iteration, we arrive at an average of 4 vgatherdps instructions per gather loop construct. From this, we can infer the runtime contribution based on our previous findings in Section 5.3.4. The latency of each gather instruction in the situation where the data is distributed across four CLs is 3.7 clock cycles. With a total of 16 vgatherdps instructions per iteration, the contribution is 59.2 clock cycles. Together with the remaining

---

2 The overhead includes the time it takes to call the line update kernel (backing up and later restoring callee-save registers, the stack base pointer, etc. onto the stack) as well as instructions in the kernel that are not part of the loop body (cf. lines 7–11 in Listing 5.9).
3 See Appendix B for the measurement methodology.
4 The value of \(4.39 \times 10^{10}\) voxels is somewhat higher than the value for Sandy Bridge shown previously in Table 4.1 on page 60. This is due to overhead caused by fixing the start and stop offsets of a voxel line to be multiples of 16, rather than multiples of 8 as was the case for Sandy Bridge.
part of one kernel loop iteration (37.5 clock cycles), the total execution time is approximately 97 clock cycles.

### 6.1.2. Cache and Memory Subsystem Contribution

To estimate the impact of the runtime spent transferring the data required for the CL update we first have to identify which transfers can not be overlapped with execution time.

Because the voxel volume is too large for the caches each CL has to be brought in from main memory for the update; eventually, the updated CL will have to be evicted as well. This means that a total of 2 CLs, corresponding to 128 byte, have to be transferred. Using software prefetching (cf. lines 69 and 70 in Listing 5.14) any latency and transferring cost from the memory and cache subsystems regarding volume data can be avoided.

Unfortunately, prefetching the projection data is not possible without serious performance penalties (a more detailed discussion on the topic of prefetching data to be gathered is provided in Chapter 7). To estimate the overhead of having to wait for the projection data we investigate various cache events with the help of counters; using this data we then deduce what fractions of the data come from the local L1 segment, the local L2 segment, and off-chip (i.e. other cores’ L2 caches and main memory).

To determine the percentage of projection data that is gathered from the local L1 cache, likwid-perfctr is used to measure the `DATA_READ_OR_WRITE` and `DATA_READ_MISS_OR_WRITE_MISS` counters. The former measures the total number of accesses (i.e. reads and writes) to the L1 cache while the latter counts the number of L1 misses.

Due to the lack of a appropriate counters, there is no direct way to measure the number of accesses to a core’s L2 segment. We can, however, use the `L2_DATA_READ_MISS_MEM_FILL` and `L2_DATA_READ_MISS_CACHE_FILL` counters—indicating the number of L2 misses that were serviced from main memory and remote L2 caches respectively—to infer the number of accesses to the local L2 segment.

All counters are measured once for an ordinary FDK kernel and again for a gather-less version of the kernel. The differences in the counters values reflect the L1 accesses and L1/L2 misses that can be attributed to the projection data.

<table>
<thead>
<tr>
<th>Version</th>
<th>L1 Accesses</th>
<th>L1 Misses</th>
<th>L2 Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Gather</td>
<td>5.63 \times 10^{10}</td>
<td>5.27 \times 10^9</td>
<td>7.92 \times 10^7</td>
</tr>
<tr>
<td>Without Gather</td>
<td>1.27 \times 10^{10}</td>
<td>2.51 \times 10^8</td>
<td>5.55 \times 10^7</td>
</tr>
<tr>
<td>Difference</td>
<td>4.36 \times 10^{10}</td>
<td>5.02 \times 10^9</td>
<td>2.37 \times 10^7</td>
</tr>
</tbody>
</table>

Table 6.1.: Number of accesses to L1 cache (`DATA_READ_OR_WRITE`), L1 misses (`DATA_READ_MISS_OR_WRITE_MISS`), and L2 misses (sum of `L2_DATA_READ_MISS_MEM_FILL` and `L2_DATA_READ_MISS_CACHE_FILL`) for reconstructions using an ordinary and a gather-less FDK kernel.

Using this data, the percentage of gathers that is serviced from L1 cache is given as $p_{PL1} =$
6. Performance Model and Validation

\[
(4.36 \times 10^{10} - 5.02 \times 10^9)/4.36 \times 10^{10} = 88.5\%.
\]
Note that the number of total L1 accesses \((4.36 \times 10^{10})\) exactly matches the number of gather instructions determined previously. The percentage of gathers serviced from main memory is \(p_{MEM} = 2.37 \times 10^7 / 4.36 \times 10^{10} = 0.05\%\). Because this value is negligible, we assume the number of gathers serviced from the L2 cache to be \(p_{L2} = 1 - p_{L1} = 11.5\%\).

The average memory subsystem contribution is thus given by the cost of bringing in 11.5\% of the gathered projection data from the local L2 cache segment. Since each gather is transferring a full CL, this amounts to approximately 16 CLs \(\times 64\) byte/CL \(\times 11.5\% \approx 118\) byte. We estimate the effective L2 bandwidth in conjunction with the gather instruction to be the following: The latency of a single gather instruction (when dealing with data that is distributed across four CLs) was previously measured to be 3.7 clock cycles with data in L1 cache, respectively 9.1 clock cycles with data in the L2 cache (cf. Table 3.4 on page 47). Assuming the difference of 5.4 clock cycles to be the exclusive L2 cache contribution, we arrive at an effective bandwidth of \(64\) byte/5.4 cycle = 11.85 byte/cycle. The average memory subsystem contribution is thus 118 byte/11.85 byte/cycle \(\approx 10\) cycles.

![Figure 6.1.](image)

Figure 6.1.: Overview of execution time and memory subsystem contribution.

Figure 6.1 provides an overview of the performance model. The upper part shows core and L1 cache, together with all data transfers from the cache. The lower part shows the memory hierarchy through which data has to be transferred to perform the line update. The blue arrows represent the CLs pertaining the voxel volume data; prefetching these CLs in time guarantees overlap of transfers with core execution. The red arrow represents the transfers of projection data which can not be prefetched; the latency of these transfers is the determining
factor for the memory subsystem contribution. This leads to a total of 107 clock cycles to perform a single CL update.

Based on the runtime of a single kernel iteration we can determine whether the memory bandwidth becomes a limiting factor for our application. For each loop iteration, 128 byte (2 CLs) have to be transferred over the memory interfaces. Each of the 60 cores is clocked at 1.047 GHz; at 107 cycles per iteration, the required bandwidth is:

\[
\frac{1.047 \text{ GHz/core}}{107 \text{ cycles}} \cdot 60 \text{ cores} \cdot 128 \text{ byte} = 70.0 \text{ GiB/s}.
\]

The required value is well below the measured sustainable bandwidth of around 165 GiB/s (cf. Fig. 3.12 on page 42), indicating that memory bandwidth is not a problem for our application.

### 6.2. Validation

Given the model, the total runtime contribution of the line update kernel is

\[
\frac{4.39 \cdot 10^{10} \text{ voxels}}{16 \text{ voxels/iteration}} \cdot \frac{107 \text{ cycles/iteration}}{60 \text{ cores} \cdot 1.048 \text{ GHz/core}} = 4.67 \text{ s}.
\]

Unfortunately, there is a non-negligible amount of time spent outside of the loop update kernel. The value obtained by measuring the runtime of the reconstruction with the call to the line update kernel commented out was 0.42s. Thus, the total reconstruction time is 5.09 seconds.

Foreclosing the runtime of the assembly implementation from the next Chapter (cf. Table 7.1), which is 5.16 seconds, we estimate the model error at 1.4%.
Chapter 7

Results and Discussion

Table 7.1 gives a summary of the performance of all versions and compares them to the refined fastrabbit implementation (cf. Section 4.2) run on the CPU reference system. All implementations were benchmarked using static OpenMP scheduling with a chunk size of 262 voxel lines—independent of the implementation, this value resulted in the best performance.

We find that all three auto-vectorization attempts result in a very limited gain when comparing them to the CPU version. The performance obtained with OpenMP’s #pragma simd was worst—being almost indistinguishable from the CPU baseline. The approach using ISPC resulted in a performance similar to that achieved with OpenMP vectorization. As far as auto-vectorization is concerned, the Intel C Compiler’s native vectorizer produced the best performance.

The results obtained with the versions written in C with intrinsics and assembly indicate that when it comes to vectorization, manual optimization is preferable as long as the focus is on performance alone. However, we would like to point out that the cost-benefit ratio increases significantly when it comes to manual vectorization: The time it took to develop the assembly kernel was roughly twice the amount required for the C with intrinsics version—the increase in performance, however, was a mere 6%. One of the reasons for this is that when trying to reduce the time spent inside the line update kernel, we can only reduce the time spent executing instructions, which only makes up about 36% of the total runtime (about 38 out of 107 clock cycles). The remaining time is spent gathering the intensity data; this portion of the line update kernel can not be improved upon, although several attempts were undertaken.

A first approach was to shuffle the vector registers that contain the offsets for the gather instruction in such a way that the left and right intensity values required for a bilinear interpolation are paired together, i.e. placed in successive components of the vector register. This way, based on a 93.75% probability the left and right offsets point to the same CL, the total number of gathers can be reduced. However, we found that the reordering was too costly instruction-wise—the benefit obtained by requiring fewer gather instructions was far

\[1\text{ A CL can hold up to 16 intensity values. There is a total of 15 position in which the left-right-pair fits into the same CL: The left intensity value in position } i, \text{ the right intensity value in position } i+1, \forall i \in [1,15]. \text{ Therefore, the probability of two successive SP floating-point numbers residing in the same CL is 15/16.} \]

93
7. Results and Discussion

<table>
<thead>
<tr>
<th>Version</th>
<th>Best [s]</th>
<th>Average [s]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>improved fastrabbit</em> on Sandy Bridge EP</td>
<td>7.86</td>
<td>7.88</td>
<td>—</td>
</tr>
<tr>
<td>Intel compiler using #pragma simd</td>
<td>7.77</td>
<td>7.84</td>
<td>1.01 ×</td>
</tr>
<tr>
<td>ISPC</td>
<td>7.66</td>
<td>7.69</td>
<td>1.02 ×</td>
</tr>
<tr>
<td>Intel compiler using -fp-model fast</td>
<td>6.99</td>
<td>7.09</td>
<td>1.11 ×</td>
</tr>
<tr>
<td>C with Intrinsics</td>
<td>5.48</td>
<td>5.54</td>
<td>1.44 ×</td>
</tr>
<tr>
<td>Assembly</td>
<td>5.16</td>
<td>5.24</td>
<td>1.50 ×</td>
</tr>
</tbody>
</table>

Table 7.1: Summary of all implementations for \( L = 512 \). The Best column shows the best performance (TTS in seconds) achieved in 25 runs, while the Average column shows the average performance achieved during the same 25 runs. The speedup in relation to the *improved fastrabbit* version for the CPU is shown in the last column, with values based on the average performance numbers.

outweighed by the additional instructions required to perform the shuffling.

Another attempt was to improve the gather L1 hit rate, thereby eliminating some of the additional latency caused by waiting for the data to be loaded from the L2 cache, by using various prefetching techniques. However, we found that the gather hint instruction, vgatherpf0hintdps, is implemented as a dummy operation—it has no effect whatsoever apart from instruction overhead. Another prefetching instruction, vgatherpf0dps, appeared to be implemented exactly the same as the actual gather instruction, vgatherdps: Instead of returning control back to the hardware context after the instruction is executed, we found that control was relinquished only after the data has been fetched into the L1 cache, rendering the instruction useless. Finally, scalar prefetching using the vprefetch0 instruction was evaluated. The problem with this approach is getting the 4 \( \cdot 16 \) offsets stored inside a vector register into scalar registers. This requires storing the contents of the vector register onto the stack and sequentially loading them into general purpose registers. Obviously, 4 vector stores, as well as 64 scalar loads and prefetches, amounting to a total of 132 scalar instructions, is too much instruction overhead. As a consequence we evaluated variants in which only every second (68 instructions), fourth (36 instructions), or eighth (20 instructions) component of the vector registers was prefetched. Nevertheless, the overhead still outweighed any benefits caused by increasing the L1 hit rate.

To summarize, we identify the gather operation as the limiting factor regarding the speedup when comparing the MIC implementations to *fastrabbit* on the CPU. While the rest of the FDK kernel can be improved by increasing the vector register width, the growing width at the same time counteracts the performance, because the cost of filling the vector registers with scattered data increases linearly with register width.
Chapter 8

Conclusion and Outlook

We have presented a detailed examination of the Intel MIC accelerator and the two-socket Sandy Bridge CPU reference system by performing various benchmarks and evaluated both platforms for their suitability to run the FDK algorithm.

We improved the original *fastrabbit* CPU implementation \(^{[14]}\) and used it as a baseline for our performance comparisons. Important considerations concerning OpenMP parallelization for the Intel MIC such as thread placement and sensible scheduling were discussed. A number of auto- and manual vectorization strategies for the Intel MIC were examined and trade offs between development time and performance discussed. Further, we identified the latency of the gather instruction to be the main bottleneck for our application: While all other parts of the FDK kernel benefited from vectorization, loading the scattered intensity values for the bilinear interpolation counteracted this effect because the cost of filling the vector registers grows linearly with the registers’ width.

In order to integrate our findings with today’s state of the art reconstruction, a comparison with the fastest currently available GPU implementation, called *Thumper* \(^{[13]}\), is shown in Table 8.1.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Version</th>
<th>Volume Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>512(^3)</td>
</tr>
<tr>
<td>2S-Xeon E5-2680</td>
<td>improved fastrabbit</td>
<td>5.6</td>
</tr>
<tr>
<td>Xeon Phi 5110P</td>
<td>Intel compiler using #pragma simd</td>
<td>5.7</td>
</tr>
<tr>
<td></td>
<td>Intel compiler using -fp-model fast</td>
<td>5.7</td>
</tr>
<tr>
<td></td>
<td>C with Intrinsics</td>
<td>6.3</td>
</tr>
<tr>
<td></td>
<td>Assembly</td>
<td>7.9</td>
</tr>
<tr>
<td>GeForce GTX 680</td>
<td>Thumper</td>
<td>8.5</td>
</tr>
</tbody>
</table>

Table 8.1: Comparison of different platforms: Results for both \(L = 512\) and \(L = 1024\) shown in GUP/s.
8. Conclusion and Outlook

As mentioned during the introduction in Section 1.2, Intel intended the Xeon Phi to compete with GPU accelerators by other vendors. However, we find that the Kepler-based GeForce GTX 680 by Nvidia can perform the reconstruction 7–8 times faster, depending on the volume’s discretization \( L \). This discrepancy cannot be explained by simply examining the platforms’ specifications such as peak Flop/s and memory bandwidth.

Two of the main causes contributing to the GPU’s superior performance for this particular application are:

i. Most computations involved in the reconstruction kernel, such as the projection of voxels onto the detector panel or the bilinear interpolation, are typical for graphics applications (which GPUs are designed for). While, due to the fused multiply-add operation, the matrix-vector multiplication is performed efficiently on both the GPU and the MIC platform, the bilinear interpolation is not. GPUs possess additional hardware called texture units, each of which can perform a bilinear interpolations using a single instruction for data inside the texture cache. In addition, the texture units can work in parallel with the CUDA cores performing the arithmetic operations adding an additional layer of parallelism in favor of the GPU. To emphasize the implications, consider that out of the total of 97 clock cycles\(^1\) for one loop iteration of the FDK kernel, 6 are used for conversion from DCS to ICS (lines 19, 21, 23, 25–27 in Listing 5.10) and 3 to weight the interpolated intensity value and update the voxel volume (line 30 in Listing 5.10, as well as lines 68 and 72 in Listing 5.14); the remaining 88 clock cycles, more than 90% of the kernel, is spent on the bilinear interpolation—which is handled by a single instruction on a GPU.

ii. Given a sufficient amount of work, Nvidia’s CUDA programming model does a better job at hiding latencies. As seen before, even in the ideal case where all data can be serviced from the L1 cache, on average, each of the gather instructions has a latency of 3.7 clock cycles. Although the Intel MIC can hide the latencies of most instructions when using all four hardware contexts of a core, 4-way SMT is not sufficient to hide latencies caused by loading non-continuous data. In contrast to SMT, Nvidia’s multiprocessors feature hardware that allows them to instantly switch between warps.\(^2\) This way, every time a warp has to wait for an instruction to complete or data to arrive from the caches or main memory, the hardware simply schedules another warp in the meantime. Given a sufficient number of warps to choose from, this approach can hide much higher latencies than the 4-way SMT in-order approach.

Although we tried to provide a comprehensive discussion on the subject of 3D reconstruction using the Intel MIC accelerator in this thesis, there remain open questions that could benefit from further investigation.

\(^1\)Using the ideal-case assumption of all intensity data being serviced from L1 cache, the empirically determined value of 4 \texttt{v gatherdps} instructions per CL, and a total of four gather loop constructs in the kernel (for top left, top right, bottom left, bottom right intensity values), the runtime of a kernel iteration is 37.5 cycles + 4 \cdot 4 \cdot 3.7 cycles \approx 97 cycles.

\(^2\)On Nvidia GPUs, the number of CUDA threads concurrently executing on a core is called warp.
For one, a detailed examination of all the factors influenced by the chunk size used in the OpenMP scheduling is still owing. While we have identified a number of contributing elements, such as the amount of L2 data sharing among hardware contexts of a core and load imbalances, the exact circumstances leading to the gradients of the graphs in Figure 5.2 remain unknown.

Although we have shown that the Intel MIC accelerator can not provide the same performance as GPUs for the task of 3D reconstruction in the interventional setting, there nevertheless might be applications that can benefit from the work in this thesis.

One promising application seems to be the reconstruction of large CT volumes. Today, the largest industrial CT scanner, which at the time of this writing is the XXL-CT device only recently installed by the Fraunhofer Institute in Fürth [31], is capable of recording projection images with a resolution of 10000×10000 pixels, corresponding to more than 380 MiB per projection image. In this setting, it is possible for main memory capacity and bandwidth to play more important roles, potentially giving CPUs, with their high memory capacities, and the Intel MIC, with its high memory bandwidth, an advantage over GPUs.

Another interesting topic of research, of course, will be to evaluate the next iteration of the Intel MIC architecture, codenamed Knights Landing, for the application once it becomes available.
Bibliography


## Appendix A

### Register Mapping in the IMCI Line Update Kernel

<table>
<thead>
<tr>
<th>Register</th>
<th>Symbolic Name(s) Referencing the Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdi</td>
<td>i</td>
</tr>
<tr>
<td>rsi</td>
<td>VOL</td>
</tr>
<tr>
<td>rdx</td>
<td>p_pad_width</td>
</tr>
<tr>
<td>rcx</td>
<td>p_float_one</td>
</tr>
<tr>
<td>r8d</td>
<td>N</td>
</tr>
<tr>
<td>r10d</td>
<td>i</td>
</tr>
<tr>
<td>zmm0</td>
<td>A0</td>
</tr>
<tr>
<td>zmm1</td>
<td>A1</td>
</tr>
<tr>
<td>zmm2</td>
<td>A2</td>
</tr>
<tr>
<td>zmm3</td>
<td>tmp0</td>
</tr>
<tr>
<td>zmm4</td>
<td>tmp1</td>
</tr>
<tr>
<td>zmm5</td>
<td>tmp2</td>
</tr>
<tr>
<td>zmm6</td>
<td>wx</td>
</tr>
<tr>
<td>zmm7</td>
<td>MM</td>
</tr>
<tr>
<td>zmm8</td>
<td>pad_width</td>
</tr>
<tr>
<td>zmm9</td>
<td>float_one</td>
</tr>
<tr>
<td>zmm10</td>
<td>u, off_br, ioff_br</td>
</tr>
<tr>
<td>zmm11</td>
<td>v, off_tl, ioff_tl</td>
</tr>
<tr>
<td>zmm12</td>
<td>w, off_tr, ioff_tr</td>
</tr>
<tr>
<td>zmm13</td>
<td>w_inv, val_bl, sval_bl</td>
</tr>
<tr>
<td>zmm14</td>
<td>ix, val_br, val_b, sval_b</td>
</tr>
<tr>
<td>zmm15</td>
<td>iy, val_tl, sval_tl</td>
</tr>
<tr>
<td>zmm16</td>
<td>iix, val_tr, val_t, val</td>
</tr>
<tr>
<td>zmm17</td>
<td>iyy, off_bl, ioff_bl</td>
</tr>
<tr>
<td>zmm18</td>
<td>w2_inv</td>
</tr>
<tr>
<td>zmm19</td>
<td>scalx</td>
</tr>
<tr>
<td>zmm20</td>
<td>scaly</td>
</tr>
<tr>
<td>zmm21</td>
<td>oscalx</td>
</tr>
<tr>
<td>zmm22</td>
<td>oscaly</td>
</tr>
</tbody>
</table>

Table A.1.: Mapping of symbolic names to registers.
Appendix B

Measuring the Number of vgatherdps Instructions in a Gather Loop

To count the number of gathers we modify each loop gather construct in our kernel to increment the contents of a register each time a vgatherdps instruction is executed (cf. Listing B.1).

```
..L100:  vgatherdps  val_bl{k2}, [I + ioff_bl * 4]
    inc    rax
    jkzd   k2, ..L101

..L101:  vgatherdps  val_bl{k2}, [I + ioff_bl * 4]
    inc    rax
    jknzd  k2, ..L100
```

Listing B.1: Counting the number of gather instructions in the FDK kernel.

After the line update kernel has finished, the rax register contains the number of gathers that were executed for a particular voxel line. With rax being the first return register (see the MIC ABI Manual [26]), this value is returned to the RCTAlgorithmBackprojection function where an OpenMP reduction clause is used to sum up the individual threads’ values to compute the total number of gathers for the reconstruction.
Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 30.10.2013
Johannes Hofmann
Matrikelnummer 21359339