Towards C-arm CT Reconstruction on Larrabee

Hannes G. Hofmann, Benjamin Keck, Christopher Rohkohl, and Joachim Hornegger

***Abstract*—Reconstruction of 3-D cone-beam CT data is a computationally complex task. Therefore, many research groups were and are currently investigating methods for hardware-acceleration. A novel many-core computing platform—code named Larrabee—is currently developed by Intel. In this work we demonstrate how the back-projection step of an FDK-based reconstruction algorithm can be implemented efficiently on Larrabee. We introduce relevant features of this upcoming hardware platform, describe how to port legacy code and show several Larrabee-specific optimizations.**

***Index Terms*—Back-projection, CBCT, CT, GPU, Larrabee, Many-core, Multi-core, Reconstruction**

TABLE I
LIST OF ACRONYMS

API	Application Programming Interface
CBEA	CELL Broadband Engine Architecture
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FOV	Field of View
FPGA	Field Programmable Gate Array
GDDR	Graphics Double Data Rate memory
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
ISA	Instruction Set Architecture
LRBni	Larrabee new instructions
SIMD	Single Instruction, Multiple Data
SP	Single Precision
SSE	Streaming SIMD Extensions
TBB	Threading Building Blocks
VPU	Vector Processing Unit

I. Introduction

Modern computing architectures offer a high level of parallelism. Current CPUs feature up to six cores per socket, each one able to process four single precision (SP) floating point numbers simultaneously. They offer a great deal of flexibility but are outperformed in terms of peak performance by modern GPUs. This is owed to the fact that GPUs hold hundreds of simple processing units. Moreover, their memory bandwidth is typically much higher due to the use of GDDR RAM. However, historically their programming model was rather stiff. Languages and frameworks for general purpose computing on GPUs (GPGPU) were developed [1]–[3] and with the advent of CUDA [4] in 2007 GPUs gained even more popularity.

Recently, Seiler et al. [5] presented details about a novel computing architecture code named Larrabee which is currently developed by Intel. Larrabee combines the best of both worlds. The add-on card consists of several CPUs and some fixed-function hardware. It can be programmed like a GPU (i.e. using DirectX and OpenGL) or be programmed natively and resemble a compute cluster on a card. More details about the architecture are given in Sect. II-A.

In an interventional environment [6] fast 3-D reconstruction of tomographic data is highly desirable. In the past, hardware acceleration of reconstruction methods was investigated by several groups, e.g. on FPGAs [7], [8], CELL [9], [10] or GPUs [11]–[13]. Currently the most wide-spread reconstruction algorithms in clinical C-arm CT belong to the class of filtered back-projection (FBP), e.g. the Feldkamp-Davis-Kress (FDK) method [14] for cone-beam data. The most time consuming part of an FBP reconstruction is the back-projection step where the volume data is actually computed. It is both computationally complex and bandwidth demanding—yet highly parallelizable. Also iterative reconstruction algorithms that contain a back-projection step would benefit from its acceleration. The outline of this paper is as follows: In Sect. II we outline the features of Larrabee. Further, we present the RABBITCT framework which we used to implement our prototypes and check their correctness. In Sect. III implementation aspects for the Larrabee programming model are introduced. It is first explained how to port legacy code. In the following, several platform-specific optimization strategies are discussed. We conclude with Sect. IV and provide an outlook of future developments.

This work was supported by Intel Corporation.

H. G. Hofmann, B. Keck, C. Rohkohl and J. Hornegger are with the Chair of Pattern Recognition, Department of Computer Science, Friedrich-Alexander University Erlangen-Nuremberg, Martensstr. 3, 91058 Erlangen, Germany. Corresponding author: Hannes G. Hofmann (e-mail: hannes.hofmann@informatik.uni-erlangen.de).

II. Methods

A. Larrabee

In the following a brief overview of the key facts and relevant new features of the Larrabee platform will be provided. A more in depth description can be found in Seiler et al. [5].

A single Larrabee core is based on the dual-issue in-order Pentium design [15]. It is augmented with a 16-wide vector processing unit (VPU) and offers support for four hardware threads and 64-bit extensions. Many of those cores are placed around a ring bus, together with fixed function hardware units and other agents like memory and I/O interfaces. Each processor has access to a 256 KB local subset of the global coherent L2 cache. The Larrabee new instructions ISA (LRBni) [16] provides some novel instructions that allow to solve problems that arise from vectorization more efficiently.

Vector masks are 16-bit unsigned integer values where each bit corresponds to an element of a SIMD vector. Most LRBni instructions have two variants. The first one operates on all vector elements equally. The second one takes a vector mask as additional parameter. Where a mask bit is set to zero, the corresponding element in the result vector is unchanged. In turn if it is set to one, the element in the result is updated. This allows to write vectorized conditional code sections in a simpler and more efficient way.

Many algorithms compute array indices on the fly and access non-contiguous memory addresses. The new gather and scatter instructions allow to load or write 16 elements from or to 16 different addresses specified in a vector register.

Most graphics cards' fixed function hardware is replaced by a software implementation on Larrabee, but it includes dedicated hardware for texture filtering. In the back-projection algorithm they can be used to efficiently access the projection images and perform a bilinear interpolation. In a ray-driven forward-projection algorithm they may be used to sample the volume with trilinear interpolation.

Actual Larrabee hardware is not available yet. Development was done in collaboration with Intel to validate results. Therefore, we were able to show the correctness of our implementations but could not estimate the performance on the future hardware.

B. RabbitCT

Once we are able to make performance estimates we want to obtain results which are comparable to other publications. Therefore, we implemented all prototypes as modules for the open reconstruction benchmark RABBITCT [17]. It provides a standardized C-arm CT dataset, well defined problem statements and a framework to support implementing and benchmarking the back-projection step of the FDK algorithm. In this paper we utilized RABBITCT to evaluate the correctness of our Larrabee implementations.

The dataset consists of $N = 496$ pre-processed projection images $\boldsymbol{I}_n \in \mathbb{R}^{S_u \times S_v}$ from a C-arm system (Siemens AG, Artis Zee) acquired on a $200°$ circular short-scan trajectory. The size of a projection image is $S_u = 1248$ pixels in width and $S_v = 960$ pixels in height. For each projection image a pre-calibrated projection matrix $\boldsymbol{A}_n \in \mathbb{R}^{3\times 4}$ is available [18], [19].

The task for RABBITCT modules is the reconstruction of an isocentric cubic volume. Three volume resolutions were chosen to represent different problem sizes with different computational costs. The side lengths of the volume are $L \in \{256, 512, 1024\}$ voxels, respectively.

III. IMPLEMENTATION

Our implementations follow the RABBITCT reference implementation. That is, the outermost loop iterates over all projections, the inner ones traverse the volume in z, y and x direction respectively. Every voxel is updated exactly once per projection.

In the first part of this section we explain how we ported our existing CPU-optimized implementation onto Larrabee and highlight the changes that were necessary. Next, we describe how we exploited specific novel features of Larrabee.

A. Enabling legacy code for Larrabee

Larrabee is x86 compatible and can run legacy code after recompiling it. However, since it is an add-on card that has its own memory some changes are required. Similar to programming of GPUs, programs are split into two parts that execute on the host (CPU) and Larrabee device respectively. Memory allocations from the device memory could theoretically be done by the device itself. However, Windows Vista's driver model requires larger allocations to be done by the host. Similar to DirectX the host program allocates buffers and adds them to a context. The device program can subsequently access them. Host and device can communicate through these buffers or use a dedicated low latency message passing API.

1) Multi-Threading: After recompiling our scalar OpenMP implementation—where we had split the z-loop—we could execute it without further changes. Moreover, an extended version of Intel's Threading Building Blocks (TBB) [20] is supported. Finally, Larrabee provides a task API with a work-stealing software scheduler. Besides OpenMP we also developed a RABBITCT module using this task API. We were able to re-use the same strategy that was used to parallelize the back-projection on CPUs using TBB.

2) Vectorization: Regarding vectorization, the first difference between Larrabee and current CPUs is the increased SIMD width. As mentioned in Sect. II-A there are also some novel SIMD instructions. Their use will be discussed in Sect. III-B since we focus on porting existing code here.

LRBni provides 512-bit versions of all SSE instructions that we used in our optimized CPU implementation. Thus we just had to do three steps to get our code compiled and running.

- Change data types to new 512-bit types
- Adjust number of loop iterations to new SIMD width
- Update intrinsics' names

The general rule for the new intrinsics' names is similar to SSE: `_mm512_<op>_<type>`, but it is not guaranteed to be the final naming convention.

B. Optimizing code for Larrabee

1) Multi-Threading: Larrabee's task scheduler is able to handle large numbers of small tasks efficiently. To exploit this we created more tasks than by just partitioning the z-loop. Therefore, we also implemented a version where we parallelized the z- and y-loops and one where all three loops were split.

Algorithm 1 shows the basic steps performed for each voxel. The processing thread stalls in line 4 when it waits for memory access. At that point processing continues with another task that is ready for execution. Given a sufficient number of tasks, main memory latency can completely be overlapped by computation.

2) Vectorization and vector masks: Vectorization is most efficient in code parts where a lot of computation is done. Therefore we used it within the x-loop where the actual back-projection happens including a bilinear interpolation. It iterates over all voxels in one column (fixed y and z coordinate) and performs the same computation on each of them. This makes it just natural to use SIMD here.

While multi-threading was supported by libraries, vectorization was done manually. For most parts of the code it was straightforward and the new multiply-add instruction were

Algorithm 1: Basic steps for back-projection of image $\boldsymbol{I}_i$.

```
1 foreach $\underline{x}$ do
2     project voxel onto detector plane;
3     (pre-)fetch projection values;
4     wait for projection values;
5     bilinear interpolation;
6     update $f_{\text{FDK}}(\underline{x})$;
7 end
```

used wherever possible. However, vectorization is particularly complicated for code sections that contain conditional branches. For example, the algorithm contains a check if the current voxel is in the Field-of-View (FOV) of the current projection before the four neighboring projection values are read. If a projected coordinate lies outside of the image the corresponding projection value should be set to zero. The interpolation and voxel update can then be performed as usual. Listing 1 shows the scalar version of the code that reads projection values.

Current SSE VPUs and GPUs cannot branch independently for individual vector elements. The projected pixel coordinates of a voxel have to be clamped to the image dimensions to avoid invalid memory access. After fetching the projection values, the outliers have to be set to 0. To implement these measures in SSE we used integer vectors as binary masks as shown in List. 2. Invalid coordinates are set to $(0, 0)$ in line 7 and in line 9 outlying pixel values are set to 0. Using the new vector masks introduced by LRBni it is possible to write the same code more concise and efficiently (lines 7 and 8 in List. 3).

3) Scatter/gather instructions: Another problem is caused by the projection geometry. Rays through adjacent voxels do not necessarily end up in neighboring pixels in the projection image. This results in non-linear memory accesses when loading the projection values for a vector of 16 voxels. Therefore, all pixel values had to be fetched in a scalar manner in SSE and inserted into vectors which were subsequently used to compute the bilinear interpolation (not shown in List. 2). Listing 3 shows how we compute the offsets of the pixels' memory addresses in a vector register (line 7) and use the new *gather* instruction to load the required values into another vector (line 8). The result vector of *gather* is initialized with `zero` and only elements which are not masked by `inside` are loaded and updated. The speedup obtained by using *gather* depends on the number of required cache lines.

4) Further Optimizations: Further performance gains could be achieved by pre-fetching values for future loop iterations or by utilizing the texture sampler hardware. While the first measure would reduce—or even eliminate—tasks stalling due to memory latency the latter one would additionally eliminate the interpolation in line 5 of algorithm 1.

IV. Conclusions

In this paper we described how to implement the back-projection step of FDK-based cone-beam reconstruction, an algorithm with clinical relevance and high demands on computation and memory bandwidth, on Larrabee. We have shown how to exploit specific features of this upcoming computing platform like many cores, wide VPUs, new instructions and fixed function hardware. Larrabee's programming model showed to be well suitable for the back-projection problem. However, we do not have performance measurements from cycle-accurate simulators or real hardware so far. Therefore, we cannot make statements about the expected performance of Larrabee for this specific task. Nor can we make judgments about the efficiency of our optimizations.

From our experience we can conclude that only little effort is required to make an existing algorithm run on Larrabee. The most notable change—for legacy code, not optimization—is the additional level in the memory hierarchy introduced by the device memory and the use of buffers to transfer data between host and Larrabee device. Porting SSE optimized and/or multi-threaded code (using OpenMP or TBB) requires only little changes if any. While the maximum speedup gained from SSE is four, the wider Larrabee VPUs justify the effort of vectorization instead of simply using more cores. However, to get best performance one must fully exploit the new features of this architecture. Manual code optimization using profiling tools is mandatory.

Other medical imaging tasks like dynamic or iterative reconstruction methods have irregular data access patterns, too. We are convinced that with its flexibility, Larrabee is well suited for many image processing tasks and can deliver significant performance improvements over existing CPU architectures. In our future research and with the availability of the first Larrabee hardware we will further investigate these issues.

Acknowledgment

The authors would like to thank Horst Haussecker and Victor W. Lee from Intel Corporation for support. Further, we want to thank the Larrabee support team.

References

[1] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: a system for programming graphics hardware in a C-like language," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*. San Diego: ACM, Jul. 2003, pp. 896–907.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*. Los Angeles: ACM, Aug. 2004, pp. 777–786.

[3] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn, "GPGPU: general purpose computation on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*. Los Angeles: ACM, Aug. 2004, p. 33.

[4] NVIDIA Corp., "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," 2007. [Online]. Available: http://www.nvidia.com/cuda

[5] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," in *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*. Los Angeles: ACM, Aug. 2008, pp. 1–15.

[6] N. Strobel, O. Meissner, J. Boese, T. Brunner, B. Heigl, M. Hoheisel, G. Lauritsch, M. Nagel, M. Pfister, E.-P. Rhrnschopf, B. Scholz, B. Schreiber, M. Spahn, M. Zellerhoff, and K. Klingenbeck-Regn, *Multislice CT*, 3rd ed. Springer Berlin Heidelberg, 2009, ch. 3D Imaging with Flat-Detector C-Arm Systems, pp. 33–51.

[7] M. Churchill, "Hardware-accelerated cone-beam reconstruction on a mobile C-arm," in *Proceedings of SPIE*, J. Hsieh and M. Flynn, Eds., vol. 6510, San Diego, Feb. 2007, p. 65105S.

```
// This snippet fetches one pixel
if (iu >= 0 && iu < S_u && iv >= 0 && iv < S_v) {
        return pI_i[iv * S_u + iu];
}
return 0.0;
```

Listing 1. Scalar version: Coordinates check, only valid coordinates are accessed, therefore no masking is required.

```
// This snippet fetches one vector of 4 pixels
// NOTE: _mm_cmp* insns:   true=>0xffffffff, false=>0x0
__m128i in_top     = _mm_cmpgt_epi32(iv, zero);          // analogous for left, right, bottom
__m128i in_v       = _mm_and_si128(in_top, in_bottom);  // analogous for in_u
__m128i inside     = _mm_and_si128(in_u, in_v);          // combine to single mask
__m128i idx        = _mm_add_epi32(_mm_mullo_epi32(iv, S_u), iu);  // compute indices
__m128i idx_masked = _mm_and_si128(idx, inside);         // set outlier indices to 0
// Not shown: load elements sequentially and insert into SSE vector "values"
__m128  val_masked = _mm_and_si128(values, inside);      // set outlier pixels to 0
return val_masked;
```

Listing 2. SSE version: Coordinates check and masking of invalid coordinates.

```
// This snippet fetches one vector of 16 pixels
const int upConv = _MM_FULLUPC_NONE;                       // no up conversion
const int scale  = sizeof(float);                          // element size for gather
__mmask in_top   = _mm512_cmpnlt_pi(iv, zero);             // analogous for left, right, bottom
__mmask in_v     = _mm512_vkand(in_top, in_bottom);        // analogous for in_u
__mmask inside   = _mm512_vkand(in_u, in_v);               // combine to a single mask
__m512i idx      = _mm512_mask_add_pi(zero, inside, _mm512_mull_pi(iv, S_u), iu);
__m512  values   = _mm512_vgatherd_loop(zero, inside, idx, (float*) pI_i, upConv, scale);
return values;
```

Listing 3. Larrabee version: Coordinates check, masking of invalid coordinates and loading of pixel values.

[8] X. Xue, A. Cheryauka, and D. Tubbs, "Acceleration of fluoro-CT reconstruction for a mobile C-Arm on GPU and FPGA hardware: A simulation study," in *Proc. SPIE*, vol. 6142, San Diego, Feb. 2006, pp. 1494–501.

[9] H. Scherl, S. Hoppe, F. Dennerlein, G. Lauritsch, W. Eckert, M. Kowarschik, and J. Hornegger, "On-the-fly reconstruction in exact cone-beam CT using the Cell Broadband Engine Architecture," in *9th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, Lindau, Jul. 2007, pp. 29–32.

[10] M. Kachelrieß, M. Knaup, and O. Bockenbach, "Hyperfast parallel-beam and cone-beam backprojection using the cell general purpose hardware," *Medical Physics*, vol. 34, no. 4, pp. 1474–1486, 2007.

[11] K. Mueller and R. Yagel, "Rapid 3D cone-beam reconstruction with the Algebraic Reconstruction Technique (ART) by utilizing texture mapping graphics hardware," *Nuclear Science Symposium, 1998. Conference Record.*, vol. 3, pp. 1552–1559, Nov. 1998.

[12] F. Xu and K. Mueller, "Real-time 3D computed tomographic reconstruction using commodity graphics hardware," *Physics in Medicine and Biology*, vol. 52, no. 12, pp. 3405–3419, 2007.

[13] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA)," in *Nuclear Science Symposium, Medical Imaging Conference 2007*, E. C. Frey, Ed., vol. 6, Honolulu, Oct. 2007, pp. 4464–4466.

[14] L. Feldkamp, L. Davis, and J. Kress, "Practical Cone-Beam Algorithm," *Journal of the Optical Society of America*, vol. A1, no. 6, pp. 612–619, 1984.

[15] D. Alpert and D. Avnon, "Architecture of the pentium microprocessor," *Micro, IEEE*, vol. 13, no. 3, pp. 11–21, Jun. 1993.

[16] Intel Corp., "Prototype Primitives Guide," 2009. [Online]. Available: http://software.intel.com/en-us/articles/prototype-primitives-guide/

[17] C. Rohkohl, B. Keck, H. G. Hofmann, and J. Hornegger, "RabbitCT – An Open Platform for Benchmarking 3-D Cone-Beam Reconstruction Algorithms," *Medical Physics*, vol. 36, no. 9, Sep. 2009.

[18] O. Faugeras, *Three-Dimensional Computer Vision (Artificial Intelligence).* The MIT Press Cambridge, Nov. 1993.

[19] K. Wiesent, K. Barth, N. Navab, P. Durlak, T. Brunner, O. Schuetz, and W. Seissler, "Enhanced 3-D-reconstruction algorithm for C-arm systems suitable for interventional procedures," *IEEE Transactions on Medical Imaging*, vol. 19, no. 5, pp. 391–403, May 2000.

[20] Intel Corp., "Threading Building Blocks," 2009. [Online]. Available: http://www.threadingbuildingblocks.org/