Bachelor Thesis - Bachelor of Computer Science

A Study of Productivity and Performance of Modern Vector Processors

Paul Springer
RWTH Aachen University
Contact: Paul.Springer@RWTH-Aachen.de

Supervised by

Prof. Dr. Martin Bücker	(Institute for Scientific Computing)
Prof. Paolo Bientinesi, Ph.D.	(Institute for Advanced Study in Computational Engineering Science)
Sandra Wienke, M.Sc.	(HPC-Group - Center for Computing and Communication)

Declaration of Independence

I hereby declare that I am the sole author of this thesis and that I only used the resources listed in the reference section.

Aachen, Germany
March 28, 2012

Paul Springer

Abstract

This bachelor thesis carries out a case study describing the performance and productivity of modern vector processors such as graphics processing units (GPUs) and central processing units (CPUs) based on three different computational routines arising from a magnetoencephalography application. I apply different programming paradigms to these routines targeting either the CPU or the GPU. Furthermore, I investigate the performance and productivity of programming paradigms such as OpenMP with respect to its auto-vectorization capabilities, Intel intrinsic AVX and Intel OpenCL for the CPU. Moreover, I examine NVIDIA's CUDA and OpenCL APIs for GPU-sided applications. The results of the performed case study yield roughly the same performances for the CPU and GPU implementations, but favour the OpenMP paradigm (i.e. the CPU) with respect to productivity.

Contents

1. Introduction

Graphics processing units (GPUs) greatly improved their performance over the last ten years. The first graphics cards have been developed in the 1990's and were targeted for the mass market. These first cards were special-purpose hardware, designed to accelerate graphic processing required in computer games. As the interest in computer games continued, GPU developers such as NVIDIA and AMD/ATI continuously improved the performance and level of parallelism of their GPUs. To this end, it became desirable to exploit this special-purpose hardware for general-purpose computations (GPGPU). Therefore, parallel programming models such as *Compute Unified Device Architecture (CUDA)* and *Open Computing Language (OpenCL)* are developed in order to utilize the hundreds of cores of modern GPUs which yield a peak performance of more than three TFLOPS (single precision) on a single GPU. These programming models greatly decreased the effort necessary to program these architectures, since they allow to utilize the GPU without the need to rewrite the application in terms of 3D-rendering APIs such as OpenGL or DirectX.

At their beginning in 2007 (CUDA) and 2008 (OpenCL), the underlying GPU architectures suffered from drawbacks such as low double-precision performance, no ECC support, few debugging capabilities and the lack of atomic operations. However, up-to-date GPU architectures such as NVIDIA's Fermi architecture and current CUDA and OpenCL implementations do not exhibit these problems anymore [16].

Today's GPUs are based on the execution model of *Single Instruction Multiple Thread* (SIMT), allowing multiple threads to process consecutive elements in lock-step. Likewise, modern CPU architectures feature the *Single Instruction Multiple Data* (SIMD) execution model which enables them to execute a single instruction on multiple data elements simultaneously. Hence, these processors operate on multiple consecutive elements concurrently and are referred to as vector-processors. This SIMD execution model adds an extra level of parallelism to the parallel execution units in recent CPUs.

Both, Intel and AMD continuously improved their vector extensions since they have been introduced in the late 1990's[1]. These continuous improvements led to Intel's *Advanced Vector Extension* (AVX) which is supported by Intel's Sandy Bridge microarchitecture and AMD's Bulldozer microarchitecture. It is intended to close the performance gap[2] between CPUs and GPUs in vector algorithms. In order to utilize these vector capabilities of CPUs there are mainly two options. First, the programmer can use the compiler's auto-vectorization capabilities which I will refer to as *implicit vectorization* and second, one can rewrite the application with explicit vector-instructions which I will refer to as *explicit vectorization*.

In order to evaluate the performance and productivity of modern vector processors, I apply different programming paradigms to the most compute-intensive routines arising from a magnetoencephalography (MEG) application. Furthermore, this thesis outlines the implementation of these routines using NVIDIA's CUDA and NVIDIA's OpenCL

[1] Intel's *MMX* in 1997 and AMD's *3Dnow!* in 1998

[2] e.g. Intel's *Core i7 Extreme* with a peak double-precision performance of 93.6 GFLOPS compared to NVIDIA's Quadro 6000 with 515.2 GFLOPS

APIs targeting the GPU and programming paradigms such as OpenMP, Intel's OpenCL and Intel's intrinsic AVX for CPU computations.

In order to elaborate on the productivity of each paradigm, I will use some metrics as described by Wienke et al. [26]:

- Learning effort: The time needed by a C/C++ programmer to learn the respective paradigm.
- Tool support: Available debuggers and libraries.
- Code expansion: Number of added or modified *source lines of code* (SLOCs).
- Code reorganization: Effort needed to reorganize the original version to fit the respective paradigm.

This study is organized as follows: Section 2 briefly describes the MEG application and its computational routines. Programming guidelines, implementations, performance and productivity results related to each paradigm targeting either the CPU or the GPU are shown in Section 3 and 4, respectively. Section 5 gives a final discussion on performance and productivity with respect to the results of this study. Section 6 concludes this thesis.

2. Magnetoencephalography Application

In this section, I briefly introduce magnetoencephalography (MEG) and describes the three most compute-intensive routines which are implemented using different programming paradigms throughout this bachelor thesis.

MEG is an non-invasive imaging technique used to display the synchronized neuronal activity within the cerebral cortex[1] of the human brain [8]. It has a high time resolution in the range of milliseconds and a spatial resolution of up to 2-3 mm for sources within the cerebral cortex [8]. The weak magnetic field induced by at least 10^{10} neurons [8] of the cerebral cortex is measured outside of the head by an array of superconducting quantum interference devices (SQUIDs). In order to deduce which part of the brain is responsible for the external magnetic field one has to solve the neuromagnetic inverse problem formulated as an unconstrained minimization problem [2], [4], [3] of the form

$$\text{Find } x \in \mathbb{R}^k \text{ such that } f(x) \longrightarrow min \tag{2.1}$$

where f is the scalar-valued objective function.

Bücker et al. [2] proposed to use a minimum p-norm approach to tackle this ill-conditioned and numerically challenging problem. Their approach, as described in [4], is designed for large-scale problems where the number of free scalar variables k, can be increased to some hundred thousands. In order to deal with the high time complexity involved in this large-scale problem, their solution is based on a subspace trust-region algorithm [6] which utilizes the first- and second-order derivatives of f for faster convergence using automatic differentiation. As we will see in the next section, these additional computations do not increase the overall complexity of the proposed algorithm (i.e. all routines belong to the same complexity-class).

The software package of Bücker et al. [3] comprises three computationally intensive routines. The first routine is responsible for the evaluation of the objective function $f : \mathbb{R}^k \longrightarrow \mathbb{R}$ which is given by

$$f(x) := \left\| \begin{bmatrix} L \\ \lambda I \end{bmatrix} x - \begin{bmatrix} b \\ 0 \end{bmatrix} \right\|_p^p \tag{2.2}$$

with $L \in \mathbb{R}^{n \times k}$, $x \in \mathbb{R}^k$, $b \in \mathbb{R}^n$, $\lambda \in \mathbb{R}$, I being the $k \times k$ identity and 0 being the zero-vector of length k. The symbol L denotes a dense *lead field* matrix (refer to [8] for detailed information), whereas n represents the number of SQUIDs used[2] and b represents the measurements of the magnetic field outside the human head. Henceforth, I will refer to this routine as the *eval* routine.

The first- and second-order derivatives of f are used by the *grad* routine and *hesse* routine, respectively. The *grad* routine evaluates the gradient $\nabla f \in \mathbb{R}^k$ for a dense

[1] the outermost layer of the brain which is 2-4 mm thick

[2] n is typically much less than k. I used $n = 128$ and $k = 512000$ as realistic numbers throughout this study.

$x \in \mathbb{R}^k$ while simultaneously evaluating the objective function $f(x)$. In comparison to the two previous routines, the *hesse* routine does not evaluate the objective function but computes the Hessian-Vector product $z = (\nabla^2 f)y \in \mathbb{R}^k$ for a given vector $y \in \mathbb{R}^k$ at a given point of interest $x \in \mathbb{R}^k$.

3. Central Processing Units (CPUs)

In the course of this section I will cover the CPU-based programming paradigms (i.e. standard C, OpenMP with auto-vectorization, Intel's intrinsic AVX and Intel's OpenCL) and their respective performances. Furthermore, in order to show the effort involved in writing high-performance applications, which can benefit from the CPU's vector capabilities, I will discuss some optimization techniques and highlight their impact on performance. Moreover, this section outlines different programming guidelines and comments on the effort needed to implement the three routines (see Section 2) for each paradigm.

Since it is crucial to understand the underlying architecture in order to write high-performance applications, this section starts with an introduction to the most salient architectural details of the used CPU architecture, Intel's Sandy Bridge microarchitecture.

3.1. Sandy Bridge Architecture

CPUs based on the Sandy Bridge microarchitecture support different levels of parallelism. On the one hand, there are multiple cores per CPU which allow an application to perform different operations on different cores simultaneously. On the other hand, there is the increasing SIMD support which allows an application to execute the same instruction to multiple data elements - per core - simultaneously. This process is referred to as *vector operation*, whereas the process of operating on a single data element at a time is called *scalar operation*. Today's Sandy Bridge processors support the *Advanced Vector Extensions* (AVX) which is an enhancement over Intel's previous vector extension called *Streaming SIMD Extensions* (SSE). CPUs with AVX support have increased the size of the registers used for vectorization from 128 bit to 256 bit, whereas the old so called *xmm* registers are an alias of the lower 128 bit of the new *ymm* registers. The increased register size allows each core to operate on up to eight single- or four double-precision variables simultaneously (see Figure 3.1). Hence, vectorization yields a theoretical peak speedup of 8× and 4× for single and double precision, respectively.

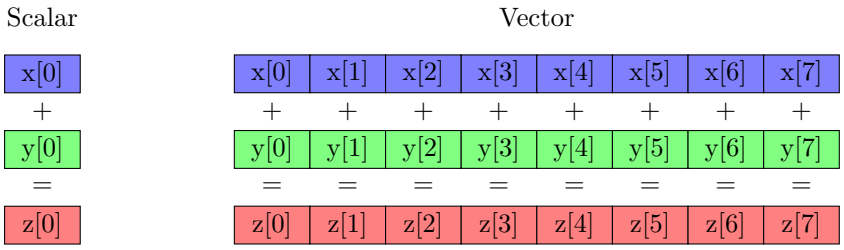


Figure 3.1: Instead of adding a single element per instruction (left), AVX allows to operate on up to eight single-precision, consecutive floating point elements simultaneously (right).

Another important property the programmer has to keep in mind is that Intel's current processor family follows the memory design of *Non-Unified Memory Access* (NUMA),

meaning that the access times to a specific memory location in main memory can differ between different cores (we will see the impact of this property in Section 3.4.3). Hence, data which is most often used by a specific core should be allocated "close"[1] to it [23].

Another feature of modern microarchitectures is their full cache hierarchy. Intel's Xeon E5-2670 for instance has a 32 KB L1 cache, a 256 KB L2 cache and a 20 MB Last Level Cache (LLC) which is shared among all cores of the CPU, whereas the L1 and L2 caches are exclusive per core. This cache hierarchy allows the programmer to exploit the locality of a given problem in order to write cache-aware algorithms which often result in higher performance (see Section 3.3.3).

With these features in mind, the programmer can achieve significant speedups.

3.2. Original C Version

The C versions of the three routines as mentioned in Section 2 form the foundation for all subsequent versions and are listed for comparison. The original C code of these routines was generated by an automatic differentiation algorithm. Hence, these routines have not been optimized to benefit from any of the aforementioned architectural features. However, I reviewed the generated code and removed unnecessary memory accesses and calculations.

3.2.1. Implementation

This section outlines the single-threaded C implementation of each routine for further references. Please note that all kernels shown in this study omit error handling and boundary checking for the sake of increased readability.

Eval Routine. The C implementation of this routine is straightforward, it is a one-to-one translation of the mathematical problem as described in equation (2.2) to standard C code. This routine can be subdivided into a dense subroutine and a sparse subroutine. Henceforth, I will refer to these subroutines as *kernels*, so that the naming is conform with the naming of such subroutines in a GPGPU sense. As the name suggests, the dense kernel (*cEvalDense*) is responsible for the computation of the dense part of equation 2.2 (i.e. $\|Lx - b\|_p^p$), whereas the sparse kernel (*cSparseEval*) computes the "sparse"[2] part (i.e. $\|\lambda x\|_p^p$) (see Figure 3.2).

[1] meaning that the access time is small

[2] meaning computationally rather simple - not to be confused with *sparse* in terms of numerical analysis

Kernel 3.2.1 cEvalSparse

Input: x, λ, p, $f(x)$
Output: $f(x)$

```
1: for j = 0 → k − 1 do
2:     tmp ← λ · x_j;
3:     f(x) ← f(x) + |tmp|^p
4: end for
```

Kernel 3.2.2 cEvalDense

Input: L, x, b, p, $f(x)$
Output: $f(x)$

```
1: for i = 0 → n − 1 do
2:     tmp ← Σ_{j=0}^{k−1} L_{i,j} · x_j;
3:
4:     tmp ← tmp − b_i
5:     f(x) ← f(x) + |tmp|^p
6: end for
```

Figure 3.2: Sparse kernel (left) and dense kernel (right) of the *eval* routine. All input and output variables are chosen accordingly to those of Section 2.

Grad Routine. While the *eval* routine is only concerned with the evaluation of the objective function f at a point of interest $x \in \mathbb{R}^k$, the *grad* routine simultaneously evaluates $z = \nabla f(x) \in \mathbb{R}^k$.

This routine can as well be subdivided into a dense and a sparse kernel (see Figure 3.3) which require $\mathcal{O}(nk)$ and $\mathcal{O}(k)$ operations, respectively. Despite the fact that these kernels incorporate small extensions (see highlighted lines of Figure 3.3) to those kernels of the *eval* routine, they still belong to the same complexity class.

Kernel 3.2.3 cGradSparse

Input: x, λ, p, $f(x)$
Output: $f(x)$, z

```
1: for j = 0 → k − 1 do
2:     tmp ← λ · x_j;
3:     f(x) ← f(x) + |tmp|^p
4:     /* Some O(1) computations */
5:     z_j ← tmp
6: end for
```

Kernel 3.2.4 cGradDense

Input: L, x, b, p, $f(x)$
Output: $f(x)$, z

```
1: for i = 0 → n − 1 do
2:     tmp ← Σ_{j=0}^{k−1} L_{i,j} · x_j;
3:
4:     tmp ← tmp − b_i
5:     f(x) ← f(x) + |tmp|^p
6:     /* Some O(1) computations */
7:
8:     for j = 0 → k − 1 do
9:         z_j ← z_j + tmp · L_{i,j}
10:    end for
11: end for
```

Figure 3.3: Sparse kernel (left) and dense kernel (right) of the *grad* routine. All input and output variables are chosen accordingly to those of Section 2. The major differences between these kernels and the kernels of Figure 3.2 are highlighted in green.

In comparison to Kernel *cGradSparse* (3.2.3), Kernel *cGradDense* (3.2.4) still offers room for further optimizations. Hence, subsequent versions of this kernel are divided into sub-kernels in order to benefit from different optimization techniques.

Hesse Routine. The *hesse* routine can once again be split into a sparse kernel (*cHesseSparse*) and a dense kernel (*cHesseDense*) which belong to the same complexity classes as those of the *eval* and *grad* routines. Moreover, these kernels only differ slightly from the kernels we have seen so far (see Figure 3.4).

Kernel 3.2.5 cHesseSparse

Input: x, y, λ, p
Output: z

```
1: for j = 0 → k − 1 do
2:     tmp ← λ · x_j;
3:     /* Some O(1) computations */
4:     z_j ← tmp · y_j
5: end for
```

Kernel 3.2.6 cHesseDense

Input: L, x, y, b, p
Output: z

```
1: for i = 0 → n − 1 do
2:     tmp_x ← Σ_{j=0}^{k−1} L_{i,j} · x_j;
3:     tmp_y ← Σ_{j=0}^{k−1} L_{i,j} · y_j;
4:
5:     tmp_x ← tmp_x − b_i
6:     /* Some O(1) computations */
7:     tmp_x ← tmp_x · tmp_y
8:
9:     for j = 0 → k − 1 do
10:        z_j ← z_j + tmp_x · L_{i,j}
11:    end for
12: end for
```

Figure 3.4: Sparse kernel (left) and dense kernel (right) of the *hesse* routine. All input and output variables are chosen accordingly to those of Section 2. The major differences between these kernels and the kernels of Figure 3.3 are highlighted in green.

3.2.2. Performance Results

The compiler used throughout this study is Intel's C compiler *icc* 12.1 using identical compiler flags (i.e. -O3, -ansi-alias, -vec-report3 and -xAVX) and running on Scientific Linux 6.1. Moreover, to provide a fair comparison of all CPU versions, the measurements are run on the same shared-memory system based on two Intel Xeon E5-2670 CPUs[1], each having eight physical cores and an LLC of 20 MB. In addition to these settings, I used Intel's compiler extension *KMP_AFFINITY* set to scatter, in order to pin the threads to specific cores. Furthermore, all subsequent versions of each routine implement roughly the same algorithm, thus it is reasonable to use metrics such as runtime and GFLOPS for a fair comparison of the different versions (i.e. there is no comparison

[1] with a peak double-precision performance of 166.4 GFLOPS

between algorithms belonging to different complexity classes). Finally, all the time measurements throughout this thesis are reproducible and taken of 100 runs using the minimum as the final result.

	Single			Double		
	eval	grad	hesse	eval	grad	hesse
Runtime	65	147	143	129	257	285
GFLOPS	2.1	1.8	2.8	1.0	1.0	1.4

Table 3.1: Runtime (in ms) and GFLOPS of each routine using either single or double precision.

The results shown in Table 3.1 indicate that all routines take roughly the same time. Moreover, we see the expected twofold drop in performance between single-precision and double-precision performance.

Further analysis of the vec-report and assembly code, generated by the compiler, shows that all kernels have been automatically vectorized.

3.3. OpenMP with Auto-Vectorization

This section describes the parallelization and vectorization of the aforementioned C versions of each routine using OpenMP [19] directives and the auto-vectorization capabilities of Intel's 12.1 C compiler.

Ideally, the programmer should not have to worry about the vectorization process and let the compiler figure out a way to vectorize the given code, this procedure is referred to as auto-vectorization. In practice, however, this is not as simple as it sounds. The programmer has to follow various programming guidelines in order to benefit from the auto-vectorization capabilities of the compiler, which I will discuss in section 3.3.1. Moreover, this section outlines the respective implementations of each routine in section 3.3.2 and analyzes the impact of various optimization techniques on the performance in Section 3.3.3.

3.3.1. Programming Guidelines

The programming guidelines described in this section apply to implicit vectorization using Intel's C compiler [9], thus this section does not list specific OpenMP guidelines. For detailed information regarding OpenMP please see [5]. Furthermore, I will discuss selected problems concerning OpenMP and vectorization in Section 3.3.3.

Intel's C compiler tries to vectorize the innermost loop of a nest and leaves the outer loops untouched. Moreover, in order to take advantage of the auto-vectorization capabilities one has to follow some fundamental guidelines:

- Avoid divergence based on the iteration.
- Determine the loop trip count at loop-entry.

- Use single entry, single exit loops (no branch in or out).
- Avoid non-contiguous memory accesses.
- Avoid indirect indexing.
- Avoid inter loop dependencies.
- Align the used data elements to 16/ 32 byte boundaries for SSE/ AVX.
- Mark the data elements to be independent of each other[1].
- Use structures of arrays (SoA) instead of arrays of structures (AoS).

For further information please refer to [9]. To get started with the auto-vectorization, I recommend to compile with *-vec-report3* which will indicate if the loops have been vectorized. In some cases *-vec-report3* will already give the reason **why** the loops have not been vectorized. Moreover, the compiler flag *-quide* can give some advice on **how** the programmer should change the program such that it can take advantage of auto-vectorization.

Even though the vec-report provides a good initial overview of the vectorization progress, it is not sufficient if the user wants to verify that the compiler completely utilized the underlying hardware. Hence, it is recommended to have a brief look at the generated assembly code to check whether the old *xmm* or the new *ymm* registers have been used and to verify that the compiler invokes the new AVX instructions instead of the old SSE instructions (see Figure 3.5). It is desirable to use the new AVX instruction (denoted by the v-prefix) and the packed version since this version operates on multiple consecutive data elements simultaneously. Please compare [11] for further information.

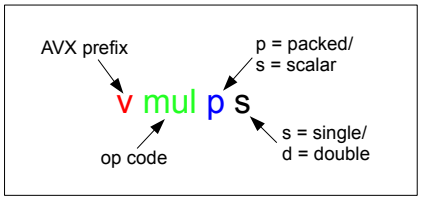


Figure 3.5: Naming scheme of an AVX assembly instruction on the example of the *vmulps* instruction.

3.3.2. Implementation

This section outlines the OpenMP implementation of each routine. The inputs (e.g. L, x) to these routines are initialized with artificial values which are chosen in a way such that the results can be reliably verified. Moreover, the initialization process is

[1] This can be done with the compiler flag *-ansi-alias*, *#pragma ivdep* in front of the loop or by using the *restrict* keyword.

distributed among all threads. This ensures that memory initialized by thread i running on core j is allocated "close" to j.

As mentioned in the previous section, data elements should be aligned to 32 byte boundaries, which allows the processor to invoke the more efficient aligned load instructions instead of the unaligned load instructions. Hence, all the data elements used in this section have an alignment of 32 byte.

Eval Routine. The parallelization of the *eval* routine is quite simple, since the kernels are pretty much the same as those of the original C version (see Figure 3.2), except for *omp for* directives in front of the loops (see Figure 3.6).

Kernel 3.3.1 ompEvalSparse

Input: $x,\ \lambda,\ p,\ f(x)$
Output: $f(x)$

```
1: # omp for reduction(+:f(x))
2: for j = 0 → k − 1 do
3:     tmp ← λ · x_j;
4:     f(x) ← f(x) + |tmp|^p
5: end for
```

Kernel 3.3.2 ompEvalDense

Input: $L,\ x, b,\ p,\ f(x)$
Output: $f(x)$

```
1: # omp for reduction(+:f(x))
2: for i = 0 → n − 1 do
3:     tmp ← Σ_{j=0}^{k−1} L_{i,j} · x_j;
4:
5:     tmp ← tmp − b_i
6:     f(x) ← f(x) + |tmp|^p
7: end for
```

Figure 3.6: Parallelization of Kernel *cEvalSparse* (left) and Kernel *cEvalDense* (right).

Grad Routine. In order to parallelize the dense kernel of the original C version and to avoid loop carried dependencies, I used two different approaches. First, I parallelized the outer loop of Kernel *cGradDense* (3.2.4) at the cost of an additional reduction[1]. Second, I split the dense kernel into two separate kernels (see Figure 3.7) using an auxiliary array $h \in \mathbb{R}^n$. I will refer to these two approaches as the *grad outer* and *grad inner* version, respectively. The sparse kernel (*ompGradSparse*), on the other hand, does not require any further attention, as it is similar to Kernel *cGradSparse* (3.2.3) with an additional *omp for* directive in front of the loop. As we will see in Section 3.3.3 both approaches yield different performances.

[1] omitted in Kernel *ompGradDenseOuter* (3.3.3)

Kernel 3.3.3 ompGradDenseOuter

Input: L, x, b, p, $f(x)$
Output: $f(x)$, z

```
1:  # omp for reduction(+:z,f(x))
2:  for i = 0 → n − 1 do
3:      tmp ← Σ_{j=0}^{k−1} L_{i,j} · x_j;
4:
5:      tmp ← tmp − b_i
6:      f(x) ← f(x) + |tmp|^p
7:      /* Some O(1) computations */
8:
9:      for j = 0 → k − 1 do
10:         z_j ← z_j + tmp · L_{i,j} /* working on local array*/
11:     end for
12: end for
```

Kernel *ompGradDenseOuter* (3.3.3) illustrates the parallelization of the outer loop (line 2). Hence the loops in line 3 and 9 can be vectorized as long as each thread is working on a local copy of z. Please note that the exact implementation of the reduction of z and the data-sharing attribute clauses are omitted due to better readability.

Kernel 3.3.4 ompGradDenseInner1

Input: L, x, h, b, p, $f(x)$
Output: $f(x)$, h

```
1:  # omp for reduction(+:f(x))
2:  for i = 0 → n − 1 do
3:      tmp ← Σ_{j=0}^{k−1} L_{i,j} · x_j;
4:
5:      tmp ← tmp − b_i
6:      f(x) ← f(x) + |tmp|^p
7:      /* Some O(1) computations */
8:      h_i ← tmp
9:  end for
```

Kernel 3.3.5 ompGradDenseInner2

Input: L, h, z
Output: z

```
1:  for i = 0 → n − 1 do
2:      # omp for
3:      for j = 0 → k − 1 do
4:          z_j ← z_j + h_i · L_{i,j}
5:      end for
6:  end for
```

Figure 3.7: Parallelization of Kernel *cGradDense* (3.2.4) using two kernels.

The parallelization of the former Kernel *cGradDense* (3.2.4) using the inner approach is outlined by Kernel *ompGradDenseInner1* (3.3.4) and Kernel *ompGradDenseInner2* (3.3.5). While Kernel *ompGradDenseInner1* is very similar to the first part of the original C dense kernel, Kernel *ompGradDenseInner2* is essentially a transposed matrix-vector product of the form $z = h^T \cdot L$. Further performance analysis showed that Kernel *ompGradDenseInner2* accounts for the major part of the overall runtime of the *grad* routine. This is mainly due to its many accesses to main memory (i.e. z is to large to be

cached). Thus, it is desirable to take advantage of the full cache hierarchy of the CPU in order to reduce memory access times as much as possible. This observation has led to Kernel *ompGradDenseInner2* (3.3.6), which exhibits much better performance (see Section 3.3.3). Because this kernel is more complex than the aforementioned kernels and the fact that I will reuse this kernel for the *hesse* routine, Figure 3.8 illustrates the control flow of this blocked version.

Kernel 3.3.6 ompGradDenseInner2Blocked

Input: L, h, z
Output: z

```
1:  numBlocks ← k/BLOCKDIM
2:  # omp for
3:  for i = 0 → numBlocks − 1 do
4:    /* Initialize tmp with 0 */
5:    for j = 0 → n − 1 do
6:      for l = 0 → BLOCKDIM − 1 do
7:        tmp_l ← tmp_l + h_j · L_{j,i·BLOCKDIM+l};
8:      end for
9:    end for
10:
11:   for l = 0 → BLOCKDIM − 1 do
12:     z_{i·BLOCKDIM+l} ← tmp_l
13:   end for
14: end for
```

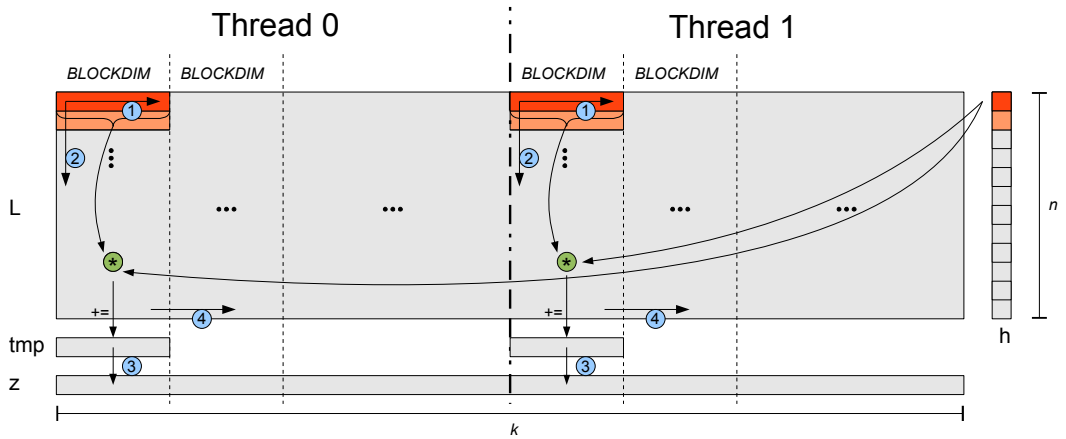


Figure 3.8: Exemplary control flow of Kernel *ompGradDenseInner2Blocked* using only two threads. The numbering within the blue cycles indicate the order of execution. Elements colored in orange denote the active elements while the elements colored in lighter orange denote the next elements. *BLOCKDIM* denotes the number of elements in each block.

Hesse Routine. The parallelization of Kernel *cHesseDense* (3.2.6) is analogous to the *grad inner* version of the *grad* routine. Hence, I split the dense kernel into two separate parts. The first part is outlined by Kernel *ompHesseDenseInner1* (3.3.7) and the second part is equivalent to Kernel *ompGradDenseInner2Blocked* (3.3.6), hence it is not listed again. Furthermore, the parallelization of Kernel *cHesseSparse* (3.2.5) does not require further evaluation, as it only requires one additional *omp for* directive in front of the loop.

Kernel 3.3.7 ompHesseDenseInner1

Input: L, x, h, b, p
Output: h

```
1: # omp for
2: for i = 0 → n − 1 do
3:     tmp_x ← Σ_{j=0}^{k−1} L_{i,j} · x_j;
4:     tmp_y ← Σ_{j=0}^{k−1} L_{i,j} · y_j;
5:
6:     tmp_x ← tmp_x − b_i
7:     /* Some O(1) computations */
8:     h_i ← tmp_x · tmp_y
9: end for
```

The calculations of tmp_x and tmp_y of Kernel *ompHesseDenseInner1* (3.3.7) in line 3 and 4 are executed in a single loop, which allows the kernel to exploit the temporal locality of the memory accesses to $L_{i,j}$.

3.3.3. Performance Results

In the course of this section, I discuss and analyze the performance of the different OpenMP versions which are outlined in the previous section. To allow a fair comparison between all CPU versions, I used the same setup as described in Section 3.2.2. Unless otherwise stated, the measurements are run on all physical cores (i.e. 16 threads). Moreover, all loops have been successfully vectorized by the compiler.

Table 3.2 makes clear that Kernel *ompGradDenseOuter* (3.3.3) yields the poorest performance. Even though the straightforward parallelization of the inner loop (see Figure 3.7) results in a performance improvement of roughly 1.4×, the blocked version (Kernel *ompGradDenseInner2Blocked*) is even faster and achieves a speedup of up to 2.4×. The improvement of Kernel *ompGradDenseInner2Blocked* over Kernel *ompGradDenseInner2* is due to the fact that all the write operations of Kernel *ompGradDenseInner2* in line 4 have to go to main memory (i.e. z is too large to be kept in the cache), while the write operations to *tmp* of Kernel *ompGradDenseInner2Blocked*, on the other hand, can be kept in the cache. Hence, the successive improvements payed off. A more detailed comparison between the blocked kernel and the non-blocked kernel of the *inner* version can be found in Table 3.3.

	Single			Double		
Kernel	Runtime	GFLOPS	Speedup	Runtime	GFLOPS	Speedup
ompGradDense-Outer	23	12.1	-	37.1	7.2	-
ompGradDense-Inner2	14.9	17.9	1.5	26.4	10.1	1.4
ompGradDense-Inner2Blocked	9.6	28.7	2.4	19.0	13.8	2.0

Table 3.2: Runtime (in ms) and GFLOPS of the OpenMP *grad* routine using different kernels. For a more detailed comparison between Kernel *ompGradDenseInner2* and Kernel *ompGradDenseInner2Blocked*, please see Table 3.3.

	Single		Double	
Kernel	Runtime	Speedup	Runtime	Speedup
ompGradDense-Inner2	9.2	-	16.2	-
ompGradDense-Inner2Blocked	6.4	1.4	9.9	1.6

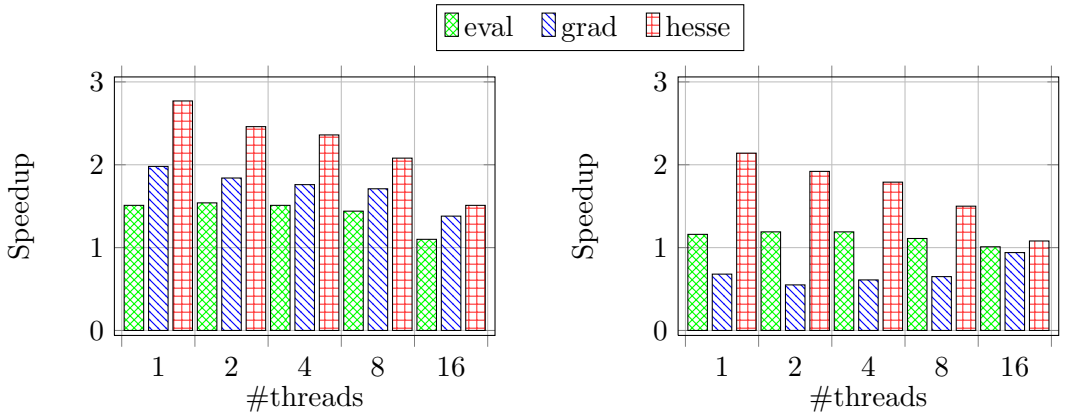
Table 3.3: Runtime (in ms) of Kernel *ompGradDenseInner2* and Kernel *ompGradDenseInner2Blocked* excluding the remainder of the *grad* routine.

In order to investigate the cooperation of OpenMP and vectorization and to demonstrate the speedup solely due to vectorization, I compiled all routines either with or without vectorization (i.e. with or without the compiler flag *-no-vec*), respectively and measured the performance for 1, 2, 4, 8 and 16 threads. The results of these measurements are illustrated in Figure 3.9.

First, Figure 3.9 illustrates that the speedup solely due to vectorization decreases with an increasing number of threads. This matches the results of my preliminary work [22]. However, the measurement of the *grad* routine using Kernel *ompGradDenseInner2Blocked* (3.3.6) seems to be an exception here. While the measurements using single precision are not extraordinary, the measurements of the *grad inner* routine using double precision are, as the speedup (due to vectorization) is increasing with an increasing number of threads. Moreover, Figure 3.9 indicates that the *grad inner* routine using double precision even suffers a significant performance decrease of up to 2× due to vectorization[1] running on 1, 2, 4 or 8 threads. A more detailed investigation of this odd behavior is left as future work.

Second, single-precision computations benefits more from vectorization than double-precision computations. This was expected since the single-precision computation allows eight elements to be processed simultaneously while double precision is restricted to four

[1] the decision of vectorization was solely up to the compiler. Hence, the compiler vectorized without being forced to



(a) Single precision

(b) Double precision

Figure 3.9: Speedup of each routine using OpenMP with auto-vectorization over the same version without auto-vectorization running on 1, 2, 4, 8 and 16 threads, respectively.

elements (see Section 3.3.1).

Furthermore, the speedup solely due to vectorization is much smaller than the theoretical peak speedups of 8× and 4× for single precision and double precision, respectively. This might be surprising because all of the computationally intensive operations are vectorized. Additionally, this seems to be even more surprising if we compare these results with the results shown by Chris Lomont (Intel) [10] who achieved speedups of up to 7.58× for single precision and 3.66× for double precision. However, a closer look at the algorithm used in [10] reveals that it only requires very few ($\mathcal{O}(1)$ many) accesses to global memory, while the routines used in this study require $\mathcal{O}(nk)$ accesses to main memory.

Finally, to demonstrate the NUMA-behavior of the CPU [23], I initialized all the data by a single thread. Hence, most of the threads encounter higher memory access times, which lead to performance degradation. A comparison between the final OpenMP versions with and without parallel initialization and the final speedups of the OpenMP routines over the original C versions are summarized in Table 3.4. Table 3.4 illustrates the importance of NUMA since the version with parallel initialization is roughly two times faster than the same version without parallel initialization. Furthermore, it shows a significant speedup of up to 16.3× for single precision and 15.8× for double precision. However, please bear in mind that these speedups are mostly due to thread-level parallelism and not due to vectorization (compare Figure 3.9).

3.3.4. Productivity

As far as the productivity is concerned, auto-vectorization can be considered to be quite easy since most of the work is automatically done by the compiler. Moreover, the kernels examined in this thesis are quite pleasant to work with when it comes to auto-vectorization as they do not require much code reorganizations. They merely require

	Single			Double		
	eval	grad	hesse	eval	grad	hesse
Runtime	4	9.6	9.1	9.3	19	18
GFLOPS	33.3	37.8	43.6	14.3	14.1	22
Speedup NUMA	2.1	1.8	1.8	2.0	1.9	1.9
Speedup C	16.3	15.3	15.7	13.9	13.5	15.8

Table 3.4: Runtime (in ms) and GFLOPS of the parallelized routines using OpenMP with auto-vectorization and parallel initialization. *Speedup C* denotes the speedup w.r.t. the original C routines (see Table 3.1), *Speedup NUMA* denotes the speedup over the same versions without parallel initialization. *Grad* denotes the *grad* routine using Kernel *ompGradDense2Blocked* (3.3.6).

the typical OpenMP directives and some additional lines of code in order to ensure the necessary alignment. It is worth mentioning that the compiler does not even require any additional compiler flags for vectorization. However, despite the fact that the vectorization process for the examined kernels is straightforward, the speedup solely due to vectorization decreases with increasing parallelization (see Figure 3.9) and is almost non-existing if 16 threads are used. However, this observation does not always hold. In fact, in my preliminary work [22], I was able to achieve speedups of $4\times$ for single precision and $2\times$ for double precision across different number of threads. Anyhow, the results shown in [22] required more code rearrangements than the kernels of this study. Furthermore, the programming guidelines are concisely presented in [9] and do not require much *learning effort*.

Additionally, OpenMP is supported by a rich set of tools. For instance:

- *Rouge Wave TotalView*: GUI-based debugger with multi-threading support and memory analysis engine.
- *Intel Inspector XE 2011*: Analyzes memory and threading behavior (e.g. memory leaks, data races, hotspots, thread level performance).

All in all it is fair to say that productivity - as a ratio of speedup over programming effort - can be considered quite high.

3.4. Intel Intrinsic AVX

Intel's intrinsic AVX gives the programmer more control over the vectorization process which can come in handy in certain situations where the compiler either is not capable of any vectorization or the result is not as fast as it could be. This extra control, however, comes at the cost of increased programming effort, because the programmer has to define the vectorization explicitly using intrinsic functions (see Section 3.4.2).

This Section gives a brief introduction to Intel's intrinsic AVX and demonstrates explicit vectorization based on Kernel *ompEvalDense* (3.3.2). The reason for choosing Kernel *ompEvalDense* is twofold. First, it is easily comprehensible. Second, as we have seen in Figure 3.9 the kernel evinces only a very small speedup due to vectorization, thus it is desirable to increase this speedup by using explicit vectorization.

3.4.1. Programming Guidelines

This section conveys the basics involved with intrinsic C functions, which are required to understand the implementation as described in Section 3.4.2. Basically, the same guidelines mentioned in Section 3.3.1 apply to intrinsics AVX as well, but instead of implicit vectorization, the programmer has to vectorize explicitly using the intrinsic functions as provided by the *immintrin.h* header file. Most intrinsic AVX functions follow the following naming scheme [10]:

__mm256_op_suffix(dataType param1, dataType param2, dataType param3)

where *__mm256* is a prefix for operations using the new 256 bit wide *ymm* registers, *op* denotes the operation code (e.g. add, mul, load) and *suffix* denotes the data type which is operated on (e.g. *ps* for packed single, *pd* for packed double, *sd* for single double). While the AVX version of the *eval* routine (see Kernel *avxEvalDense* (3.4.1)) only uses the *__mm256d* data type, which denotes 256 bit as four double-precision floating point values, there exist several other data types as well. Please see [10] for more detailed information.

3.4.2. Implementation

The AVX implementation of Kernel *ompEvalDense* (3.3.2) is essentially a one-to-one translation to AVX intrinsics. Nevertheless, the AVX implementation (see Kernel *avxEvalDense* (3.4.1)) is suited to illustrate the collaboration between OpenMP and AVX and to demonstrate the explicit vectorization in a comprehensible way (see Figure 3.10).

```
1 #pragma omp parallel for private(i,j) reduction(+:funval)
2 for(i = 0; i < n; ++i)
3 {
4     __m256d tmpRow, chunkOfL, chunkOfX, chunkOfLX;
5     tmpRow = _mm256_broadcast_sd(&zero);                    //STEP 1
6
7     for(j = 0; j < k; j+= 4)
8     {
```

```
9           chunkOfL   = _mm256_load_pd( &L[i*k+j] );          //STEP 2
10          chunkOfX   = _mm256_load_pd( &x[j] );              //STEP 3
11          chunkOfLX  = _mm256_mul_pd( chunkOfL, chunkOfX );  //STEP 4
12          tmpRow     = _mm256_add_pd( chunkOfLX, tmpRow );   //STEP 5
13      } //STEP 6
14      double tmp[4];
15      _mm256_store_pd( tmp, tmpRow );
16      tmp[0] = tmp[0] + tmp[1] + tmp[2] + tmp[3] - bhat[i];
17      result += pow(fabs(tmp[0]), p);
18  } //STEP 7
```

Kernel 3.4.1: *avxEvalDense*. Implemention of Kernel *ompEvalDense* (3.3.2) using Intel's intrinsic AVX functions. Each step in this kernel works on four double values simultaneously and is illustrated in 3.10.

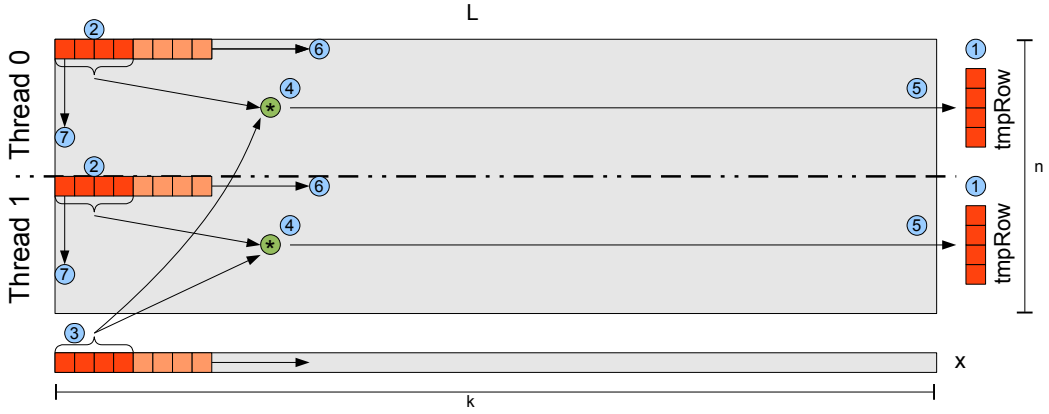


Figure 3.10: Exemplary control flow of Kernel *avxEvalDense* using only two threads. The steps shown in this figure are aligned to the steps shown in Kernel 3.4.1. Elements colored in orange denote the active elements while the elements colored in lighter orange denote the next elements.

As it is depicted in Figure 3.10, each thread works on its own set of rows. Hence, there are no dependencies between the threads and the steps are as follows:

1. Initialize tmpRow with zeros.
2. Load four consecutive double values of L to chunkOfL.
3. Load four consecutive double values of x to chunkOfX.
4. Multiply chunkOfX and chunkOfL element-wise.
5. Add the result to tmpRow.
6. Return to step 2 until the end of the row is reached.
7. Return to step 1 until all rows have been processed.

3.4.3. Performance Results

Time measurements reveal, that there is no performance difference (i.e. speedup is 1) between Kernel *avxEvalDense* using explicit vectorization and Kernel *ompEvalDense* (3.3.2) using implicit vectorization. Even though this is not the result I was hoping for, it shows the good auto-vectorization capabilities of Intel's C compiler.

3.4.4. Productivity

In this particular case, Intel intrinsics seem to be a poor choice because the explicit vectorization does not only require additional knowledge about Intel's intrinsic AVX functions but also requires additional programming effort. Nevertheless, AVX intrinsics give the programmer more control over the vectorization process which can be necessary in certain situations.

However, there exists an alternative to the intrinsics, namely *user-mandated* vectorization (using *#pragma simd*) [9]. User-mandated vectorization can be seen as a compromise between full control (intrinsics AVX) and little control (auto-vectorization). It is based on compiler directives which allow explicit vectorization of the subsequent loop. The interested reader is referred to [9] for further information.

The vectorization in terms of intrinsics AVX was straightforward and took only little time using the OpenMP kernel as the origin. Nevertheless, I recommend to use auto-vectorization or user-mandated vectorization first and shift the attention to intrinsics if the speedup is not satisfactory.

3.5. Intel OpenCL

This section covers the implementation and analysis of the three routines of Section 2 using Intel's OpenCL SDK 1.5 running on Scientific Linux 6.1. Intel's OpenCL SDK 1.5 fully conforms with the OpenCL 1.1 specification [7]. Please note that I refer to the CPU as the *host* and to the device which is responsible for executing the kernel as the *device*. This distinction might be unnecessary, since the device and the host in this section are the same (i.e. the CPU), however, this distinction is conform to the terminology which I use in Section 4.

OpenCL is an open standard for programming heterogeneous systems (e.g. consisting of CPUs and GPUs). One idea behind OpenCL is to increase portability and to reduce the effort needed to take advantage of GPUs as co-processors. Thus, it is desirable to write the application once and choose between different compute devices (e.g. NVIDIA GPUs, AMD/ATI GPUs or CPUs) as available. Hence, ideally it should be possible to run the same program on the CPU or GPU without the need of extensive code changes, just by specifying the desired compute device. Even though this works in some cases, it typically results in performance degradation because different compute devices require different optimization techniques (see Table 3.5).

OpenCL follows the programming scheme of *Single Program Multiple Data* (SPMD). A kernel (single program) is executed concurrently, whereas each instance works on different data elements (multiple data). These kernel instances are referred to as **work-items**.

The number of work-items is determined by an N-dimensional **index space** (see Figure 3.11) which is explicitly specified by the programmer or implicitly determined by the OpenCL implementation. Moreover, the index space is partitioned into **work-groups**, which consist of multiple work-items. The *execution model* of OpenCL is outlined in Figure 3.11.

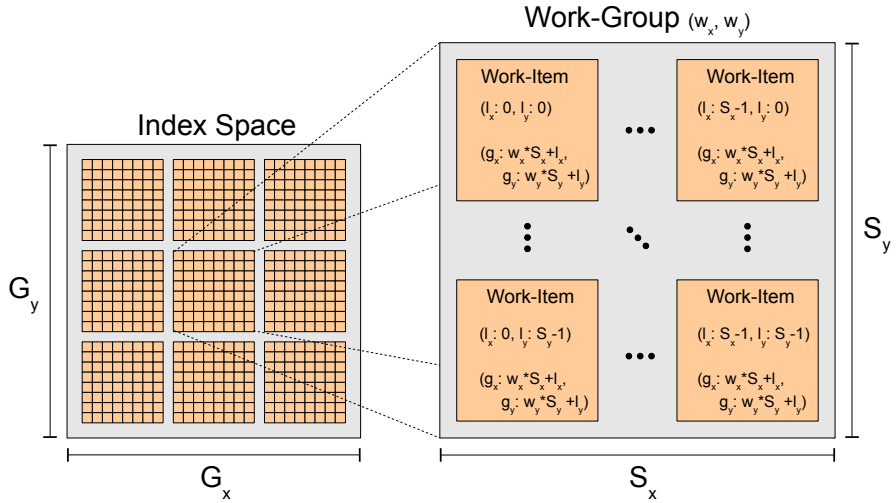


Figure 3.11: OpenCL Execution Model: The *index space* consists of $G_x \cdot G_y$ many work-items, while each work-group consists of $S_x \cdot S_y$ many. Each work-group has its unique id within the *index space* (w_x, w_y). Moreover, each work-item has an unique local id (l_x, l_y) within its work-group and an unique global id (g_x, g_y) within the *index space*. In the style of [13].

The execution model is mapped onto OpenCL's *platform model*, which consists of **compute devices** (CD), **compute units** (CU) and **processing elements** (PE) (see Figure 3.12). Multiple work-groups can be scheduled to a single CU, whereas multiple work-items are mapped to a single PE. Furthermore, each CD has a **global memory** and each CU has its own **local memory**.

Different work-groups can access the global memory independently of each other. Work-items belonging to the same work-group can collaborate and ...

- ... share data through local memory.
- ... synchronize execution using barriers and memory fences.
- ... utilize special work-group functions[1].

However, work-items belonging to different work-groups can not exploit these features. Hence, work-groups are meant to be independent of each other.

[1] e.g. functions to prefetch data from global memory

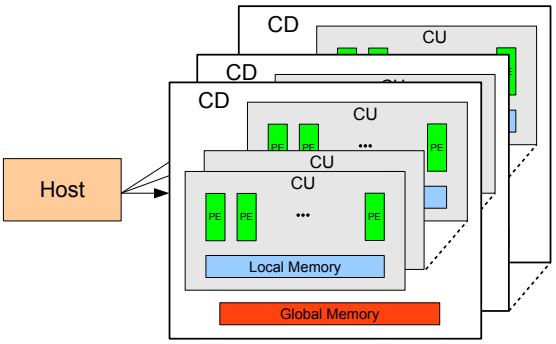


Figure 3.12: OpenCL Platform Model: The host can control multiple CDs, whereas each CD can consist of multiple CUs which in turn consists of multiple PEs.

3.5.1. Programming Guidelines

This section focuses on the basic techniques and programming guidelines unique to Intel's OpenCL SDK for the CPU and neglects general optimization techniques for OpenCL. For further information regarding OpenCL in general, please refer to [7] or [15]. Furthermore, I will outline the most salient differences between OpenCL on CPUs and OpenCL on GPUs.

Intel's OpenCL SDK is able to utilize the SIMD capabilities of the underlying CPU, in a way it is supporting two different forms of vectorization, namely *implicit* vectorization and *explicit* vectorization. Since implicit vectorization is done by the compiler, the programmer does not have to do any additional work. Explicit vectorization, on the other hand, uses built-in vector data types (e.g. float8, double4) such that each work-item operates on multiple consecutive data elements in lock-step. Thus, the programmer is responsible for changing the index space size accordingly (i.e. reduce the size w.r.t. the vector width). Explicit vectorization can become beneficial if the implicit vectorization turns out to yield bad performance and one wants to hand-tune the respective kernel. Moreover, in order to benefit from the vectorization capabilities of the CPU, one should pay attention to the same programming guidelines as those in Section 3.3.1.

The data type used in the kernel should match the register width of the underlying architecture (e.g. float8 for AVX and float4 for SSE). However, the compiler might still decide to merge consecutive work-items if it seems beneficial. On the other hand, using larger vector types than supported by the CPU is comparable to loop-unrolling this might improve the performance at the cost of increased register pressure and scales for future CPUs.

Another important property is the proper choice of the work-group size. This is a complex task, as it is influenced by many factors. First, since work-groups are independent of each other, they can be mapped to different hardware threads. Thus, it is recommended to have at least as many work-groups as physical cores. Second, increasing the number of work-groups will increase the amount of parallelism, which gives the OS

more freedom with respect to scheduling. Having to many work-groups, on the other hand, reduces the number of work-items per work-group, which in turn increases the runtime overhead. A solution might be to introduce "heavy" kernels (i.e. kernels with a larger workload), which will amortize the runtime overhead. In order to exempt the programmer from this decision, it is possible to let the compiler choose the right work-group size. Even though this might be a good choice in some situations, I encourage the programmer to experiment with different work-group sizes as they can have a severe impact on performance (see Figure 3.15).

Furthermore, programmers used to write OpenCL applications for GPUs should pay attention to the following guidelines, when writing OpenCL applications for CPUs:

- Avoid needless memory transfers.
- Avoid the usage of local memory, since all memory access are cached anyway and the *__local* qualifier only introduces an additional overhead.
- Avoid event based synchronization because these calls block the underlying hardware thread. Using functions such as *clFinish*, on the other hand, allow the thread to participate in kernel execution.

For further information on optimization techniques regarding Intel's OpenCL SDK, please refer to [13].

The introduced programming guidelines in this section improved the performance at the cost of decreased portability (e.g. using host memory for memory accesses within the kernel might lead to performance degradation while running on GPUs).

3.5.2. Implementation

An OpenCL application can be roughly broken down in the following steps:

1. Create the computing context.
   a) Choose the compute device.
   b) Compile the kernel for that device.
   c) Create kernel objects.
   d) Create a command queue[1].
2. Transfer the data to the device.
3. Launch the kernel.
4. Transfer data back to the host.

[1] each CD can have multiple command queues

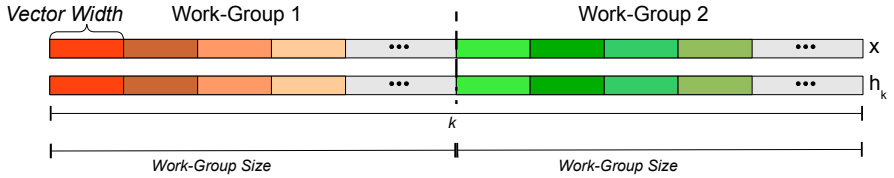
At the time of implementing the Intel OpenCL versions, I had already completed the work on the OpenMP and GPU OpenCL versions (see Section 4). Thus, I started out by using the GPU OpenCL versions and chose the CPU as the compute device. Even though this approach did not consume much time, it did not yield the desired performance of the Intel OpenCL versions (see Table 3.5). Moreover, since the GPU versions have been optimized for the GPU (see Section 4.4), it would have required much effort to modify these GPU versions. Thus, I decided to rewrite the kernels in an OpenMP-like manner. This turned out to be a troublesome work as well, but resulted in good performance, as we will see in the upcoming section.

Additionally, most of the subsequent versions are similar to their OpenMP counterparts. Thus I will not give implementation details but illustrate the work distribution among the work-items and work-groups. Moreover, all compute-intensive parts are computed on the device side (i.e. using OpenCL), whereas the less compute-intensive parts (i.e. reductions) are computed on the host side. The reason for this is that device sided reductions require synchronization across different work-groups, which is not directly supported by OpenCL and would require a new kernel invocation. In addition, since I use the CPU, there are no data transfers necessary, thus there is no benefit in reducing the temporal results on the device in order to reduce memory transfers. This, however, is different with GPUs (see Section 4.4).

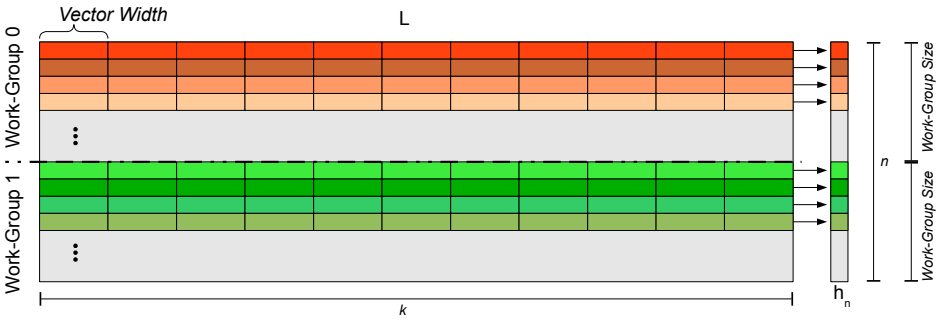
Eval Routine. The implementation of this routine is quite similar to its OpenMP and intrinsic AVX counterparts. There are essentially two kernels, a dense kernel (*cpuOCLEvalDense*) and a sparse kernel (*cpuOCLEvalSparse*). An exemplary distribution of work-items belonging to two work-groups of Kernel *cpuOCLEvalDense* and Kernel *cpuOCLEvalSparse* is depicted in Figure 3.13. While Figure 3.13 only shows two work-groups, the real implementation uses at least 16 work-groups in order to fully utilize the underlying architecture. Moreover, both kernels are using explicit vectorization by utilizing the built-in vector data types. Hence, each work-item works on four double-precision or eight single-precision elements simultaneously (indicated by *vector width* in Figure 3.13).

In order to compute the final value of $f(x)$, the temporal arrays h_n and h_k in this figure are reduced using OpenMP on the host side. Since, work-groups can be mapped to hardware threads, these versions are more or less equal to the OpenMP version (compare Kernel *ompEvalSparse* (3.3.1) and *ompEvalDense* (3.3.2)), thus I assume similar performance results.

Grad Routine. Similar to the *eval* routine, the *grad* routine is roughly the same as its OpenMP counterpart which uses the blocked version for its dense computations (i.e. Kernel *ompGradDenseInner2Blocked* (3.3.6)). Hence, this implementation can be divided into a sparse kernel (*cpuOCLGradSparse*) and two dense kernels (*cpuOCLGradDense1* and *cpuOCLGradDense2*). While the workload distribution of Kernel *cpuOCLGradSparse* and Kernel *cpuOCLGradDense1* is similar to the distribution of Kernel *cpuOCLEvalSparse* and Kernel *cpuOCLEvalDense* of the previous section, the work



(a) cpuOCLEvalSparse



(b) cpuOCLEvalDense

Figure 3.13: Exemplary work distribution of kernel *cpuOCLEvalSparse* (a) and *cpuOCLEvalDense* (b) among two work-groups and their work-items. The work assigned to each work-item is colored differently.

distribution of Kernel *cpuOCLGradDense2* requires further evaluation.

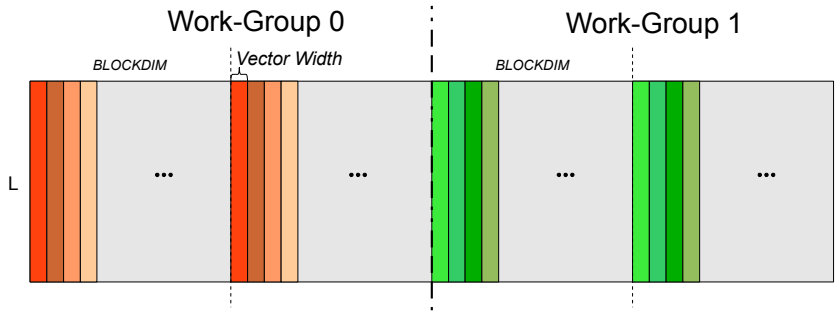
Since the control flow of Kernel *cpuOCLGradDense2* is equivalent to the control flow of Kernel *ompGradDenseInner2Blocked* (3.3.6) it is not listed again (compare Figure 3.8). The straight-forward way to divide the work among the work-items following an OpenCL-like style would be to remove the outer loop by starting as many work-items per work-group as the loop trip count (i.e. *BLOCKDIM*[1] many work-items, see Figure 3.14 (a)). This results in light-weighted kernels. Figure 3.14 (b), on the other hand, illustrates the work distribution of a very heavy-weighted kernel. This approach uses only one work-item per work-group and is very similar to the OpenMP implementation (see Kernel *ompGradDenseInner2Blocked* (3.3.6)). However, this feels counter-intuitive when writing OpenCL applications.

Performance analysis of these two versions shows that the latter approach following the OpenMP-like style is approximately two times faster than the OpenCL-like approach. Henceforth, I will refer to the OpenMP-style implementation when talking about performance of this routine.

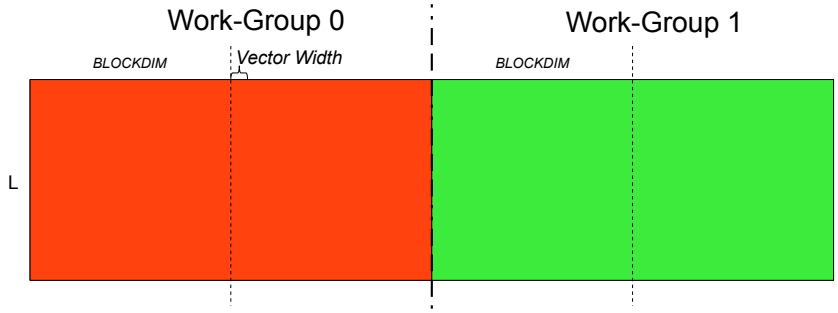
Furthermore, both variants are vectorized explicitly via the built-in vector data types.

Hesse Routine. Analogous to the OpenCL routines mentioned before, the OpenCL implementation of the *hesse* routine is similar to its OpenMP counterpart (i.e. it is split into three kernels). The kernels and their work distribution among the work-groups,

[1] *BLOCKDIM* is set to 128



(a) OpenCL Style



(b) OpenMP Style

Figure 3.14: Exemplary work distribution of Kernel *cpuOCLGradDense2* among two work-groups and their work-items. The work assigned to each work-item is colored differently. The *vector width* denotes the number of elements which are operated on simultaneously. This value is the same for both variants (a) and (b).

however, does not require further evaluation as it is analogous to the work distribution of the *grad* routine. Moreover, all kernels of the *hesse* routine are explicitly vectorized as well.

3.5.3. Performance Results

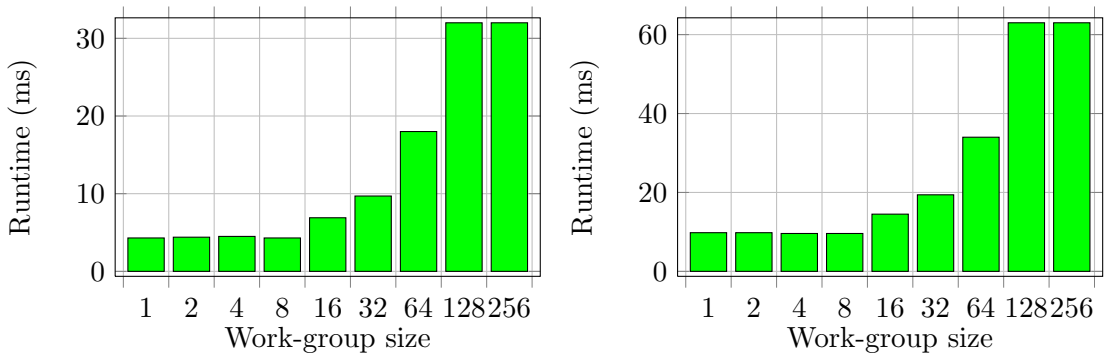
The performance results shown in this section are gathered using the same measuring environment as in Section 3.3.3. Moreover, the results neglect the building overhead of the OpenCL context, since this is a minor part of the overall runtime and is amortized over several application runs[1].

[1] i.e. it can be completely avoided by using pre compiled binaries

Figure 3.15 illustrates the exemplary effect of the work-group size on the runtime of the *eval* routine. With regard to the *eval* routine, we can derive the following:

1. Work-group sizes less or equal to 8 result in the same performance.
2. Work-group sizes larger than 8 increase the overall runtime.
3. Work-group sizes larger than 128 result in the same performance.

In order to explain these phenomena, please recap that I distributed the rows of the matrix $L \in \mathbb{R}^{n \times k}$ among the work-items (i.e. $n = 128$). Furthermore, each work-group can be mapped to a hardware thread. Hence, work-group sizes less or equal to 8 result in at least $128/8 = 16$ work-groups. Furthermore, work-group sizes larger than 8 result in less work-groups than there are physical cores available. Hence, the underlying hardware is not fully utilized. Given these performance results, it is obvious that this parameter needs to be finely tuned to the application and hardware architecture. An alternative is to leave this choice up to the compiler. In this case, however, the compiler decided to use a work-group size of 128 which results in the worst performance.



(a) Single precision

(b) Double precision

Figure 3.15: Runtime of the *eval* routine using Kernel *cpuOCLEvalDense* and different work-group sizes.

Another important feature of OpenCL is its portability between different compute devices. Table 3.5 shows the speedup of the Intel OpenCL versions as outlined in the previous section (i.e. specially optimized for the CPU) over the NVIDIA OpenCL versions (i.e. specially optimized for the GPU) of Section 4, both running on the CPU. Please note, that the GPU versions are not adapted in order to fit to the CPU. Hence, they do not even use parallel initialization which could result in a potentially twofold speedup. Nevertheless, the NVIDIA OpenCL versions would still be roughly two times slower than the optimized CPU versions. This demonstrates the necessity of tuning the versions for the applied compute device in order to achieve high performance.

A final performance comparison between the OpenCL versions running on the CPU and their OpenMP and C counterparts is outlined in Table 3.6. The results show that

	Single			Double		
	eval	grad	hesse	eval	grad	hesse
Speedup	3.5	5.8	4.6	2.4	3.1	2.7

Table 3.5: Speedup of the Intel OpenCL routines (see Section 3.5.2) over the NVIDIA OpenCL routines (see Section 4.4), both running on the CPU.

Intel's OpenCL versions yield almost the same performance as the OpenMP versions. The slight performance differences might be due to an additional OpenCL overhead.

	Single			Double		
	eval	grad	hesse	eval	grad	hesse
Runtime	4.3	9.8	13	9.6	21	25.4
GFLOPS	31.0	27.3	30.5	13.9	12.7	15.6
Speedup C	15.1	15.0	11.0	13.4	12.2	11.2
Speedup OMP	0.9	1.0	0.7	1.0	0.9	0.7

Table 3.6: Runtime (in ms) and GFLOPS of OpenCL routines discussed in this section. *Speedup C* and *Speedup OMP* denote the speedups with respect to the performance of the serial C routines (see Table 3.1) and the improved OpenMP versions (see Table 3.4), respectively.

3.5.4. Productivity

In comparison to the OpenMP versions, the OpenCL versions require an additional overhead due to the OpenCL context creation. Even though the process of OpenCL initialization is tedious, it is not a huge drawback since it can be reused for future OpenCL applications with little effort. The more severe drawback however, is the limited *tool support*. The only debuggers available are Graphic Remedy's gDEBugger and Intel's OpenCL SDK Debugger. While gDEBugger is a stand-alone debugger supporting different operating systems (e.g. Microsoft Windows 7, Linux and Mac OS X 10.6), Intel's OpenCL SDK Debugger, on the other hand, is only available as a Microsoft Visual Studio 2008 plug-in.

Further useful tools for Intel's OpenCL SDK are the Intel Offline Compiler and Intel's VTune Amplifier XE 2011. The Offline Compiler can be used to verify whether a kernel has been vectorized and to inspect the generated assembly code. However, I recommend to use VTune for such analysis since it can show the generated assembly code with respect to profiling information or the source code. For example, it allows the user to inspect the assembly code of hotspots[1]. For further information regarding Intel's tool support for OpenCL, please refer to Intel's OpenCL User Guide [12].

Additionally, even though Intel's OpenCL SDK achieves comparable performances to OpenMP (see Table 3.6) the effort needed to achieve these results is much higher than

[1] portions of the code that require much time

using OpenMP directly. Furthermore, OpenCL required the work-group size to be finely tuned, which needed some extra work.

Nevertheless, the GPU versions run on the CPU immediately results in a performance increase of 2.4× to 5.6× over the serial C versions (see Table 3.5 and 3.6), hence OpenCL on the CPU might be a good choice in situations where the GPU implementation already exists or portability is required[1]. However, if one wants to write CPU-based applications, I recommend to use OpenMP + auto-vectorization, since it is a more intuitive way to program for a CPU and offers a better tool support than Intel's OpenCL.

The effort involved in learning the most important optimization techniques for Intel's OpenCL API is very low, since it is concisely explained in Intel's documentation [13]. Nevertheless, additional knowledge of OpenCL is required. Hence, I consider the *learning effort* for a C/C++ programmer (who has never used OpenCL before) relatively high.

Finally, I point out that AMD offers the *AMD Accelerated Parallel Processing SDK* (APP) which incorporates OpenCL support for AMD's GPUs and CPUs alike. I am planing to investigate APP with respect to its performance and productivity in the near future.

[1] these speedups do not include any optimizations tailored for the CPU

4. Graphics Processing Units (GPUs)

This section describes the GPU-side implementations of the routines of Section 2 using NVIDIA's APIs for the Compute Unified Device Architecture (CUDA) and OpenCL. Since the CUDA and OpenCL implementations of the different routines are essentially the same, I do not distinguish between CUDA and OpenCL in Section 4.4. Nevertheless, Section 4.5 shows the performance results for CUDA and OpenCL separately.

The graphics card used throughout this section is NVIDIA's Quadro 6000[1], which is based on NVIDIA's Fermi architecture. Section 4.1 and 4.2 describe the Fermi architecture and the execution model of NVIDIA's CUDA and OpenCL APIs, respectively. To show the effort involved in writing high-performance GPU applications and to summarize a subset of the applied optimization techniques, I outline some programming guidelines in Section 4.3.

Furthermore, in order to avoid the new terminology that comes with CUDA, I will restrict to the OpenCL terms, which I have introduced in Section 3.5.

4.1. Fermi Architecture

NVIDIA GPUs based on the Fermi architecture consist of up to 512 cores (PEs) which make these architectures well suited for data-parallel applications. The PEs are organized in multiple CUs and can perform floating point or integer operations. A Fermi graphics card comprises a global memory of up to 6 GB GDDR5 and up to 16 CUs (compare with the OpenCL platform model (see Figure 3.12)). The global memory is accessible by all CUs, which in turn consist of 32 PEs (i.e. 512 PEs in total), *special function units* (SFUs)[2], a configurable on-chip local memory of 48 KB/ 16 KB, a configurable L1 cache of 16 KB/ 48 KB, 32768 registers á 4 byte and further units for scheduling and memory accesses (see Figure 4.1). Moreover, a Fermi GPU is capable of running up to 1536 work-items per CU simultaneously. Hence, all work-items running on the same CU need to share the available resources, this leads to little storage per work-item. However, work-items can circumvent this limitation by using the global memory at the cost of higher memory latencies and lower bandwidth. Modern GPUs offer various solutions, which can compensate this disadvantage, such as prefetching, task level parallelism or a rich variety of memories.

Prior to any computation, the required data has to be transferred from the host to global memory that resides on the graphics card (i.e. the device). These transfers often limit the use of GPUs for applications which require much data to carry out few computations.

Even though global memory offers a bandwidth of up to 177 GB/s, which is quite high compared to 51.2 GB/s of DDR3-1600 (quad-channel) of the host, it is shared by all work-items and becomes a limiting factor for some high-performance applications [1]. For instance, if 448 work-items run on an NVIDIA's Quadro 6000 GPU simultaneously,

[1] Performance of 1030.4 GFLOPS/ 515.2 GFLOPS for single precision/ double precision.
[2] Used to accelerate functions such as sin(), cos(), exp(), log(), etc. at the cost of numerical precision.

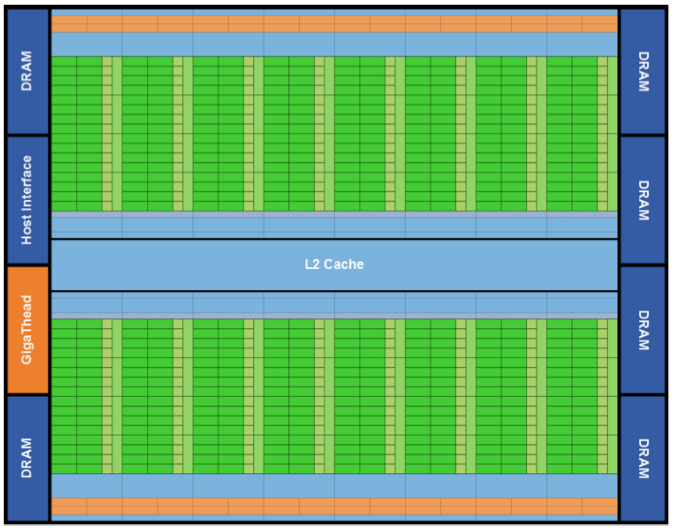


Figure 4.1: Fermi architecture: 16 CUs positioned around a common L2 cache. Each CU is depicted as a vertical rectangular strip that contains an orange portion (scheduler and dispatcher), a green portion (PEs), a light green portion (SFUs) and light blue portions (register file and L1 cache). Taken from [16].

each working on different single-precision data elements, they would require a global-memory bandwidth of $448 \cdot f_{PE} \cdot 4$ byte$/cyclesPerInstruction = 1030.4$ GB/s [1] in order to avoid idling cores due to the limited memory bandwidth. To hide the high latency and the low memory bandwidth of global memory, it is often necessary to have a *high compute to global-memory access ratio* (CGMAR) [14], [20].

Fermi is NVIDIA's first GPU architecture which tackles this problem by introducing an L2 cache of size 768 KB which can cache all accesses to global memory (additionally to the user-managed local memory). The full cache hierarchy of Fermi is a further step towards general-purpose computing. It can improve the performance of applications such as sparse linear algebra, sorting, ray tracing, fluid physics applications, and many more [21].

4.2. Execution Model

In the course of this section, I discuss the execution model of NVIDIA's CUDA and OpenCL APIs on the example of OpenCL. Nevertheless, all information apply to CUDA and OpenCL alike.

NVIDIA's OpenCL API is analogous to the execution model from Section 3.5. Hence, a single program is executed by multiple work-items simultaneously which are organized in work-groups. In comparison to Intel's OpenCL API, NVIDIA divides each OpenCL

[1] $f_{PE} = 1.15$ GHz and $cyclesPerInstruction = 2$

work-group into so called *warps* (i.e. 32 consecutive work-items).

Warps within the same work-group are scheduled on the same CU, which enables work-items belonging to the same work-group to share data by using the on-chip local memory and further collaborate by synchronizing their execution via light-weighted barriers. The warp-scheduling is done by a warp scheduler, which is able to efficiently schedule the warps, since the whole execution context (e.g. program counter, registers) is kept on the CU throughout the lifetime of the warp [18].

Moreover, NVIDIA OpenCL employs the *Single Instruction Multiple Thread* (SIMT) architecture, meaning that all work-items belonging to the same warp execute the same instruction in lock-step. Work-items within the same warp are still allowed to issue different operations (i.e. follow different execution paths). However, this results in serialization of the different execution paths.

Furthermore, global-memory accesses are issued by a warp as well. For instance, if all the work-items of a warp access consecutive words (four consecutive bytes) in global memory, OpenCL is capable of coalescing these 32 accesses to one global-memory access[1].

Local memory is divided into 32 banks, each storing multiple words. Local-memory accesses of work-items belonging to the same warp can be coalesced under certain circumstances as well. For details, refer to [18].

Concerning correctness, the programmer does not have to worry about the SIMT-behavior (execution-path divergence, memory coalescing, ...) of the GPU, but it becomes crucial to achieve good performance. This is comparable to the caching and vectorization capabilities of today's CPUs, which do not affect the correctness but can improve the performance.

4.3. Programming Guidelines

The guidelines mentioned in this section apply to NVIDIA's CUDA and OpenCL APIs alike.

Since multiple warps are executed on one CU, they need to share the available resources such as registers and local memory. Even though these resources are quite large, they become scarce if many work-items run simultaneously. For instance, if 48 warps[2] (i.e. 1536 work-items) run simultaneously, each work-item has only 21 registers to work on. However, if more than 21 registers are required by each work-item, the number of concurrently running warps on a single CU is reduced (the same holds for local-memory usage). As a result, the available parallelism decreases. Hence, the usage of shared resources needs to be taken into account to utilize the underlying hardware as good as possible. For further information regarding an effective use of shared resources, please see [24].

In order to overcome the low global-memory bandwidth and high latency, it is of high priority to leverage the much faster, on-chip local memory of the CU (for performance results see Figure 4.7).

[1] Different GPU architectures have different coalescing requirements [18].
[2] This is the maximum number of concurrently running warps on a single CU.

As mentioned in the previous section, memory-access coalescing for global and local memory is another important feature of NVIDIA's GPUs. For instance, it is possible to coalesce multiple memory accesses to local memory to one memory access, if all accesses can be served by different banks (see Figure 4.2). On the other hand, work-items accessing different words within the same bank have to be serialized (i.e. they suffer an n-way serialization penalty). The impact of bank conflicts on performance is shown in Figure 4.7.

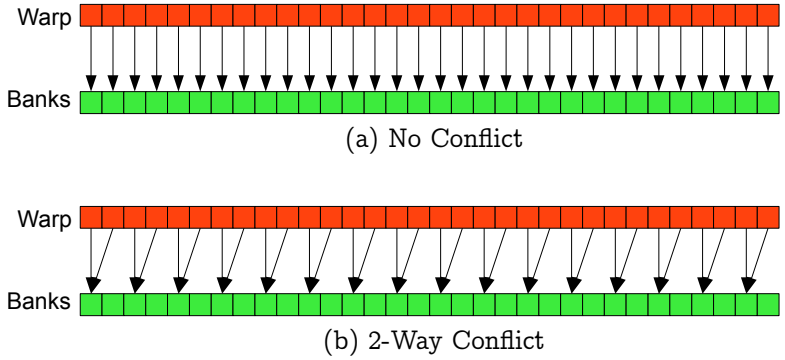


Figure 4.2: Exemplary accesses to local memory. The work-items belonging to the same warp are highlighted in orange , while the local-memory banks are highlighted in green. (a) 32 accesses to different banks, resulting in only one coalesced memory access. (b) 32 accesses to different words within 16 banks resulting in 2 memory accesses.

Execution-path divergence is yet another limiting factor for high-performance GPU applications. Even though work-items within the same warp are allowed to follow different execution paths, it would result in an n-way performance decrease. Hence, divergence within the same warp should be avoided. Execution-path divergence of work-items belonging to different warps, on the other hand, does not imply performance penalties (e.g. the reduction operations outlined in the upcoming section pay attention to this property).

For further optimization techniques regarding GPU applications, refer to [17] or [20].

4.4. Implementation

This section outlines the GPU implementations of the routines of Section 2. While it does not show code snippets, it still conveys the control flow and work distribution among the work-groups. Furthermore, due to the fact that the sparse kernels are essentially equal to the CPU kernels and the fact that they have only minor impact on the runtime, I omit detailed descriptions about the sparse kernels.

Additionally, all compute-intensive loops within the kernels are unrolled. We will see the performance implication of loop-unrolling in Section 4.5.

Eval Routine. In the course of this section, I discuss the implementation of a more advanced algorithm of the *eval* routine, which is suited to show the effect of local-memory usage and bank conflicts (for a naive GPU implementation, please see Appendix A.1).

In comparison to the CPU versions, the GPU implementation of the *eval* routine can be split into three kernels. A sparse kernel (*gpuEvalSparse*), a dense kernel (*gpuEvalDense*) and a reduction kernel (*gpuEvalReduce*) which reduces the temporary result of the kernel *gpuEvalDense*. Moreover, Kernel *gpuEvalSparse* and Kernel *gpuEvalDense* (along with Kernel *gpuEvalReduce*) are queued to different command-queues, which enable the warp scheduler to interleave their execution and keep the GPU busy.

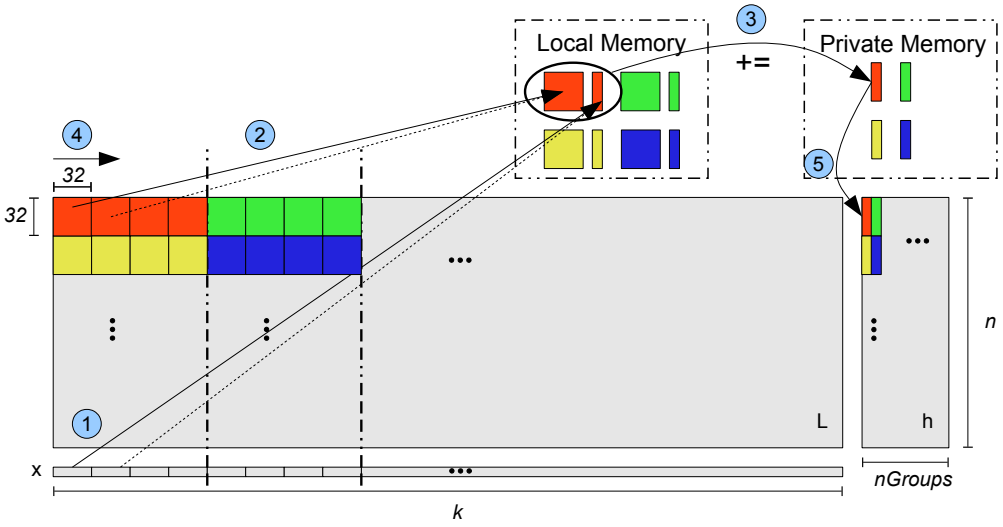


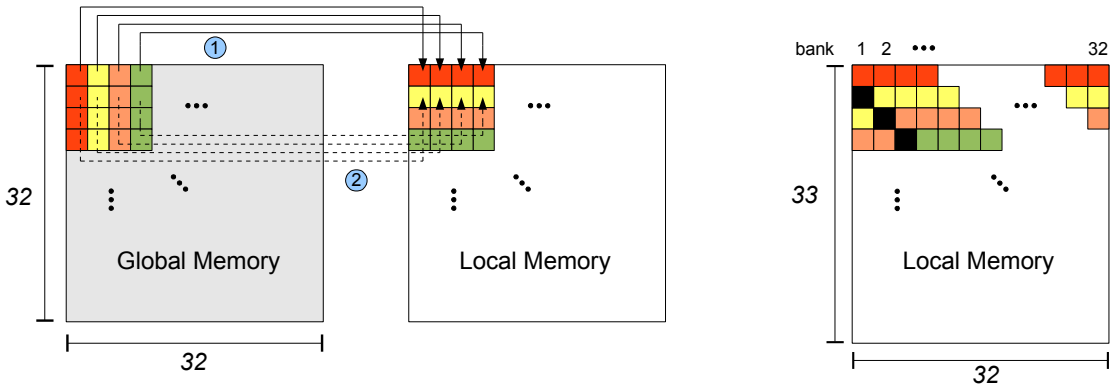
Figure 4.3: Exemplary control flow and work distribution of Kernel *gpuEvalDense*. Each color denotes a different work-group. The numbers indicate the control flow of each work-group. *nGroups* denotes the number of work-groups per row.

The control flow and workload distribution of Kernel *gpuEvalDense* is illustrated in Figure 4.3. Each work-group starts by loading its chunks of L and x from global memory to local memory (step (1) and (2)). The process of fetching the chunks of L and x has to be carefully designed in order to utilize Fermi's coalescing capabilities as good as possible. Hence, consecutive work-items load consecutive data elements of L and x. This process is illustrated in Figure 4.4 (a), it shows that all work-items cooperate in order to load the chunk of L to local memory. Thus, each load of 32 consecutive elements in a row of L can be served by one coalesced memory access for single precision and two coalesced accesses for double precision. On the other hand, if I had neglected the coalescing requirements and accessed L with a stride of k (matrix is stored in row-major order), it would have required 32 independent memory accesses and have decreased the effective memory bandwidth by a factor of 32 or 16 for single precision or double precision, respectively. Section 4.5 shows the performance difference between these two approaches.

After loading the chunks to local memory, each work-group starts to process its own

32×32 chunk of L, stores the result to its private memory (step (3)) and continues with the next chunks of L and x (denoted by the dashed arrows, step (4)). After each work-group has processed all its chunks of L, the final result is stored to the auxiliary array h (step (5)). More precisely, each work-item within the same work-group operates on its own row of the local-memory chunk (i.e. 32 consecutive elements). Following this execution scheme, however, causes a 32-way bank conflict (refer to Section 4.3). A solution to this problem is to increase the number of columns by one (padding), such that all elements which are required during the same step reside in different memory banks (i.e. no bank conflicts) (see Figure 4.4 (b)). The effect of bank conflicts on performance is shown in Figure 4.7.

Moreover, this procedure allows all work-items within the same work-group to reuse x. However, this version still suffers from two main disadvantages: First, x is redundantly loaded by work-groups processing the same columns of L and second, the extensive local-memory usage reduces the number of concurrently running work-groups per CU. An optimized version of this kernel is outlined in Appendix A.2.



(a) Loading Scheme (b) Padding

Figure 4.4: Local-memory loading scheme (a). Padding (b), avoids bank conflicts. Different colors denote the relevant elements for different work-items within the same work-group. Black elements denote the padding.

Another advantage of this algorithm is that all work-groups operate independently of each other, which results in high parallelism and enables the GPU to hide memory latencies.

Kernel *gpuEvalReduce* reduces the auxiliary array $h \in \mathbb{R}^{n \times nGroups}$ of Kernel *gpuEvalDense* to $h_{reduced} \in \mathbb{R}^n$. This approach has two main advantages. Firstly, the reduction can benefit from the massively parallel GPU architecture and secondly, the amount of data that needs to be transferred from the device to the host is reduced by a factor of *nGroups*.

The simplified[1] work distribution and control flow of Kernel *gpuEvalReduce* is illus-

[1] the real implementation uses several warps per row with a warp-size of 32

trated in Figure 4.5. The execution scheme is as follows:

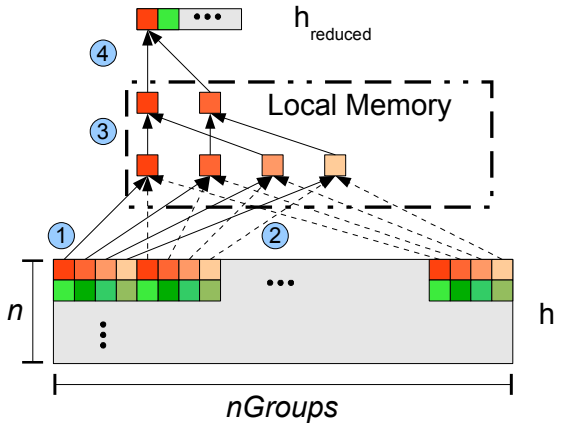


Figure 4.5: Exemplary work distribution and control flow of Kernel *gpuEvalReduce* with an exemplary warp-size of 4. The orange colored elements denote different work-items belonging to the same work-group (green analogous). Each row-sum is computed by a single work-group.

1. *Step 1 + 2*: Each work-item computes its local sum.
2. *Step 3*: The result is reduced in local memory per work-group.
3. *Step 4*: The final result is stored to $h_{reduced}$.

This algorithm pays attention to work-item divergence, since work-items belonging to the same warp follow the same execution path as long as possible.

Finally, the final sum-reduction of $h_{reduced} \in \mathbb{R}^n$ is computed by the host, since the computation on the device would require an additional kernel-invocation which does not exhibit sufficient parallelism.

Grad Routine. This routine can be split into one sparse kernel (*gpuGradSparse*), one reduction kernel (*gpuGradReduce*) and two dense kernels (*gpuGradDense1* and *gpuGradDense2*). The control flow and workload distribution of Kernel *gpuGradDense1* and Kernel *gpuGradReduce* is similar to Kernel *gpuEvalDense* and Kernel *gpuEvalReduce*, respectively.

Kernel *gpuGradDense2* is a transposed matrix-vector product (i.e. $h^T_{reduced} \cdot L$)[1] and is comparable to Kernel *ompGradDenseInner2Blocked* (3.3.6) and Kernel *cpuOCLGradDense2* (see Figure 3.14 (a)).

The workload distribution and control flow of Kernel *gpuGradDense2* with respect to its GPU implementation is outlined in Figure 4.6. The kernel launches several (i.e. $\lceil k/BLOCKDIM \rceil$) independent work-groups. Each work-item operates on its own

[1] $h_{reduced}$ denotes the auxiliary array computed by Kernel *gpuGradReduce*

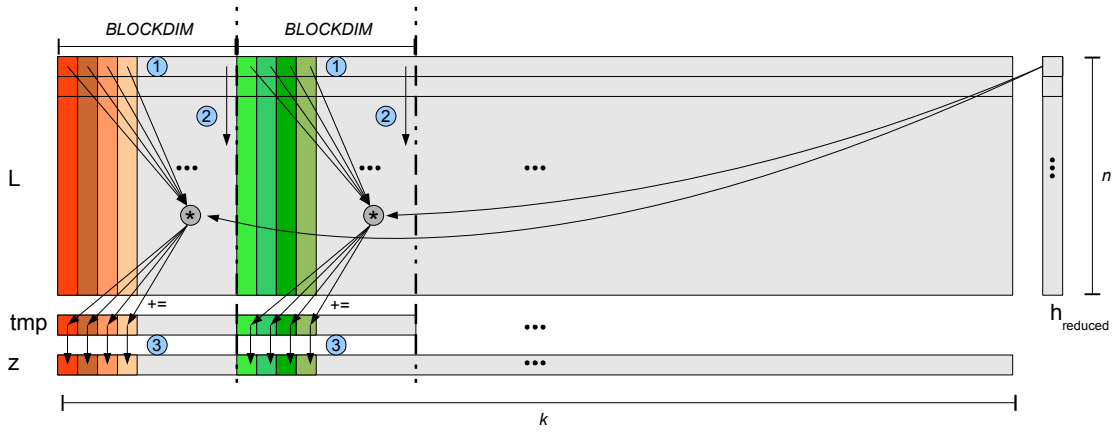


Figure 4.6: Control flow and workload distribution of Kernel *gpuGradDense2*. *BLOCKDIM* denotes the number of work-items within a work-group. Different colors denote the responsibilities of different work-items.

column. Since all work-items within a warp operate on successive elements, all accesses to global memory (i.e. accesses to L) can be coalesced. Hence, in comparison to the CPU implementations, the GPU implementation does not suffer a performance penalty for accessing the matrix L with a stride of k. Moreover, each work-item uses the private[1], auxiliary variable *tmp* for intermediate results. The final result of each column is added to z in step (3).

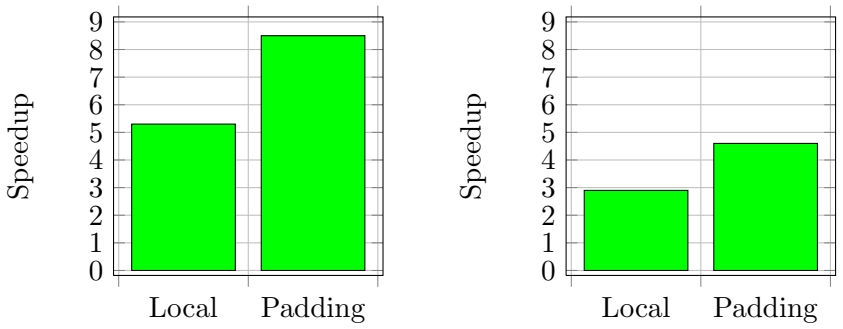
Hesse Routine. This routine is split into four kernels as well. One sparse kernel (*gpuHesseSparse*), one reduction kernel (*gpuHesseReduce*), and two dense kernels (*gpuHesseDense1* and *gpuHesseDense2*). All of these kernels contain only slight modifications to the kernels of the GPU implementation of the *grad* routine. Hence, they are not explicitly listed again.

4.5. Performance Results

All kernels are run on an NVIDIA Quadro 6000 using either OpenCL 1.1 or CUDA 4.0 (compiler flags: *-O3* and *-arch_sm=20*[2]). The host-sided computations are executed by Intel's Xeon X5650 processor (codename *Westmere*). They are single threaded and only account for a very small fraction of the runtime (roughly 0.5%).

Moreover, to provide a fair comparison between the CPU and the GPU, all time measurements include all memory transfers[3] between host and device, except for the transfer of the matrix L, since L is constant for several hundred kernel invocations of the MEG application.

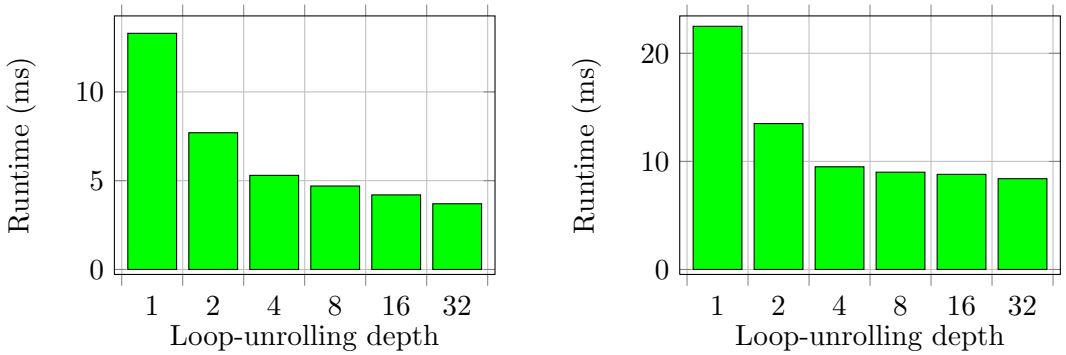
[1] only visible to the work-item
[2] only used with CUDA
[3] memory transfers account for roughly 10-20% of the runtime



(a) Single precision (b) Double precision

Figure 4.7: Kernel *gpuHesseDense1* using CUDA. Speedup due to *local-memory usage* and local-memory usage in combination with *padding*.

All dense kernels make extensive use of the local memory. Figure 4.7 shows the impact of the usage of local memory and padding on the runtime of Kernel *gpuEvalDense* using CUDA (OpenCL behaves similarly). As mentioned earlier, the theoretical bandwidth increase due to local-memory usage would be 32× for single precision and 16× for double precision. Hence, it is not surprising, that the double-precision computations benefit less from local-memory usage than the single-precision computations. Moreover, padding resolves bank conflicts and increases the performance even further. Nevertheless, the performance increase is not even close to the theoretical speedups. This could be due to Fermi's L1 and L2 caches, which cache accesses to global memory in these runs. A further explanation is the high parallelism of the kernels, which enables the architecture to hide high memory latencies.



(a) Single precision (b) Double precision

Figure 4.8: Kernel *gpuHesseDense1* using CUDA. Impact of loop-unrolling on the runtime.

Another important property is task-level parallelism and prefetching. Loop-unrolling

is one technique which can be used to exploit these properties. Hence, all dense kernels make use of loop-unrolling. Figure 4.8 illustrates the performance increase of Kernel *gpuHesseDense1* using different loop-unrolling depths. Further analysis revealed the following characteristics:

1. Speedup due to loop-unrolling is substantial.
2. Register pressure increases with increasing loop-unrolling depth.
3. OpenCL is more modest in terms of register usage than CUDA.
4. CUDA and OpenCL compilers can optimize more aggressively, if the *loop trip count* is known at compile-time.
5. Double-precision calculations require more registers than single-precision calculations[1].

To be more precise, CUDA requires 62 registers at a *loop-unroll depth* of 32, while OpenCL uses only 22 registers[2]. Even though this behavior does not result in performance implications for the analyzed kernels, it could easily become a limiting factor for some high-performance applications, since all work-items, running on the same CU, need to share the available resources. Therefore, extensive use of registers could decrease the number of parallel running work-groups on one CU.

For instance, Kernel *gpuHesseDense1* runs 32 work-items per work-group. Furthermore, Fermi provides 32768 4-byte registers, thus the register usage of CUDA would restrict the number of concurrently running work-groups to $\lfloor 32768/(62 \cdot 32) \rfloor = 16$, while OpenCL would allow $\lfloor 32768/(22 \cdot 32) \rfloor = 46$ work-groups to run concurrently. Even though this difference is substantial and would allow the warp scheduler to hide memory-access latencies more efficiently, it does not lead to performance implications for Kernel *gpuHesseDense1*. This is due to the fact, that the number of concurrently run work-groups is not limited by the register pressure but by its extensive local-memory usage. For instance, Kernel *gpuHesseDense1* uses 4352 byte of local memory using single precision (8704 byte for double precision), which restricts the number of concurrently running work-groups to $\lfloor 49152/4352 \rfloor = 11$ (5 work-groups for double precision)[3] . In practice, however, only 8 work-groups per CU can run simultaneously. This would allow each work-item to use $\lfloor 32768/(8 \cdot 32) \rfloor = 127$ registers. According to [24], CUDA imposes a soft-limit of 63 registers per work-item, which limits the use of prefetching and task-level parallelism.

Table 4.1 concludes this section. It shows that CUDA and OpenCL yield roughly the same performances. A final comparison between the performance of the GPU and the CPU is given in Section 5.1. Moreover, a performance comparison between Kernel *gpuEvalDense* and NVIDIA's *cuBLAS* library is outlined in Table 5.1.

[1] a double-precision element requires two registers, while a single-precision element only requires one register

[2] this observation holds for all *loop-unrolling depths*

[3] each CU has 49152 byte of local memory

		Single			Double		
		eval	grad	hesse	eval	grad	hesse
CUDA	Runtime (ms)	4.8	8.6	8.7	10.6	17.8	17.9
	GFLOPS	27.7	31.1	45.6	12.6	15.0	22.2
OpenCL	Runtime (ms)	4.9	9.3	9.9	9.8	17.9	19.6
	GFLOPS	27.2	28.7	40.1	13.6	14.9	20.2

Table 4.1: Runtime and GFLOPS of routines discussed in this section using either CUDA or OpenCL.

4.6. Productivity

OpenCL and CUDA roughly require the same programming effort (i.e. the kernels look almost the same). The only notable difference is the context creation of OpenCL.

Moreover, the *learning effort* involved in writing high-performance applications with OpenCL and CUDA with respect to NVIDIA's Fermi architecture is roughly the same, since both offer more or less the same functionalities and good documentations. Still, OpenCL is more verbose than the CUDA runtime API and requires additional knowledge (e.g. explicit context creation). However, this additional *learning effort* is negligible.

As far as tool support is concerned, both CUDA and OpenCL offer tools such as debuggers and profilers. The most notable tools are:

- *Graphic Remedy gDEBugger*: Debugger and performance analyzer for OpenCL.
- *NVIDIA parallel Nsight*: Debugger[1] and performance analyzer for CUDA.
- *NVIDIA Visual Profiler*: Performance analyzer for CUDA and OpenCL.
- *Rouge Wave Totalview*: Debugger for CUDA.
- *Allinea DDT*: Debugger for CUDA.
- *AMD APP*: Performance analyzer for AMD's GPUs.

Additionally, both CUDA and OpenCL offer rich sets of available libraries (e.g. BLAS, FFTs) which utilize the GPU. However, in comparison to CUDA[2], OpenCL still lacks an LAPACK implementation.

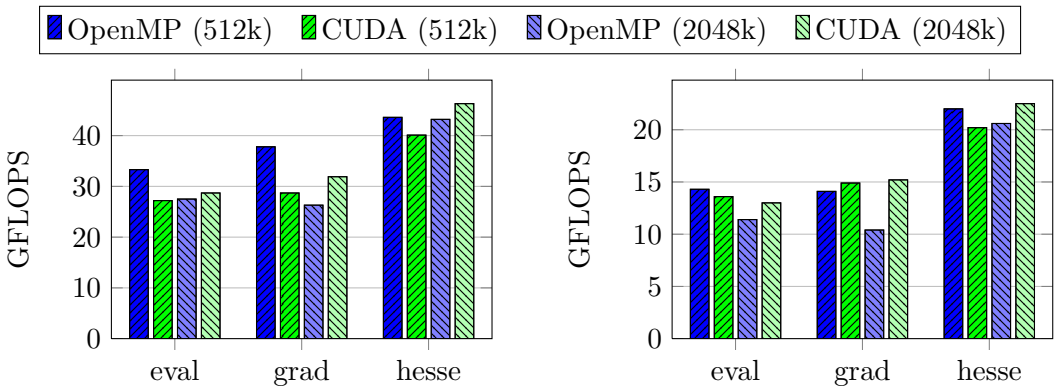
[1] only for CUDA
[2] CUDA offers two LAPACK implementations: MAGMA and CULA

5. Comparison GPUs and CPUs

This section gives a final comparison between the productivity and performance of the CPU implementations and GPU implementations of Section 3 and Section 4.

5.1. Performance

Figure 5.1 shows the performance results of the CPU implementations and the GPU implementations side-by-side for different problem sizes using either OpenMP or CUDA.



(a) Single precision (b) Double precision

Figure 5.1: GFLOPS of the implemented routines using either CUDA or OpenMP with respect to different problem-sizes. *512k*/ *2048k* denotes the size of the matrix $L \in \mathbb{R}^{n \times k}$ with $k = 512000$/ $k = 2048000$.

The results show that the performance differences between the CPU and GPU implementations is small across all routines. However, the OpenMP implementations[1] outperform the CUDA implementations for a smaller problem-size, while the GPU implementations trump the CPU implementations for a larger problem-size. Additionally, as it is demonstrated in Appendix A.2, the GPU versions still offer room for further optimizations.

Moreover, the double-precision performance is roughly half of the single-precision performance for CPU and GPU implementations alike, this shows the improved double-precision capabilities of today's graphics cards.

In comparison to the results outlined in this work, the performance results shown by Bordawekar et al. [1] and Wienke et al. [26] favour the GPU over the CPU (performance difference of 2× to 9×, depending on the used architecture). The salient difference between their results and the results of this study are due to different *compute to memory access ratios*. For instance, the algorithms used in this work require $\mathcal{O}(nk)$ memory accesses to carry out $\mathcal{O}(nk)$ computations, while the computations in [1] perform $\mathcal{O}(n^4)$ operations on $\mathcal{O}(n^2)$ data elements.

[1] the *grad* routine using double-precision is an exception here. A similar behaviour of this routine is outlined in Figure 3.9. Further investigation of this anomaly is left as future work.

Additionally to the performance results outlined so far, Table 5.1 presents the performance results of Kernel *ompEvalDense* [1] (CPU) and Kernel *gpuEvalDense* [1] (GPU) using single precision in comparison to an *sgemv*[2] operation using either Intel's *Math Kernel Library* (MKL) or NVIDIA's *cuBLAS*[3] library. The results indicate that the implementations outlined in this study outperform the implementations of well known libraries by a factor of roughly 13× or 25× for the CPU or GPU, respectively.

	ompEvalDense	MKL	*gpuEvalDense*	cuBLAS
Runtime	3.3	44	3.7	94

Table 5.1: Runtime (in ms) of Kernel *ompEvalDense* and *gpuEvalDense* compared to a *sgemv* operation using either MKL (CPU) or cuBLAS (GPU). All computations are using single precision and a matrix L of size 128×5120000.

Further analysis showed that even though the *MKL* uses all physical cores, it does not utilize the vectorization capabilities and does not benefit from parallelization (i.e. the performance using 1 thread and the performance using 16 threads is almost the same). Moreover, the poor performance of the *cuBLAS* implementation is most likely due to the structure of the matrix (i.e. L has only 128 rows but 512000 columns). More precisely, *cuBLAS* only launches 128 work-items, which is not enough in order to saturate NVIDIA's Fermi GPU.

5.2. Productivity

As an quantitative attempt to measure the productivity, Figure 5.2 illustrates the number of modified or added SLOCs for each routine using different programming paradigms with respect to the original C version. Figure 5.2 only accounts for SLOCs directly related to the routines (i.e. SLOCs related to operations such as OpenCL context-creation or verification of the results are not counted). OpenMP was the only paradigm which essentially allowed to reuse the original code. This is due to the fact that OpenMP is a directive-based paradigm, which allows incremental parallelization.

The main reasons for the high number of SLOCs for OpenCL and CUDA are:

- Computations have to be wrapped in kernels.
- Appropriate index space sizes have to be determined.
- Memory needs to be allocated.
- Memory needs to be transferred between host and device[4].

In addition, OpenCL requires the kernel arguments to be specified via explicit functions. Hence, the OpenCL implementations yield yet another overhead over the CUDA

[1] matrix-vector multiplicaion with small computational overhead
[2] single precision general matrix-vector multiplication
[3] CUDA implementation of BLAS
[4] does not apply to Intel OpenCL

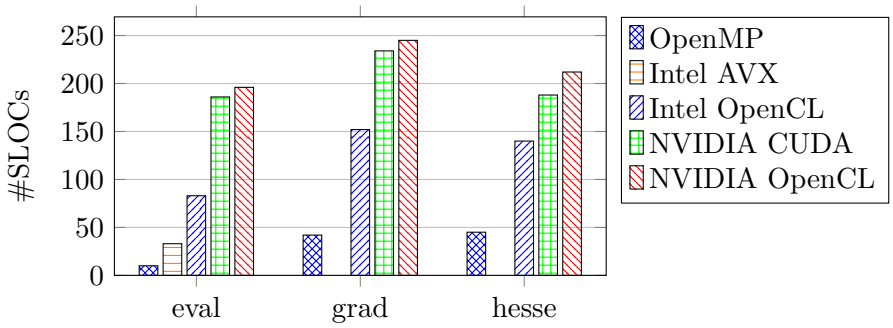


Figure 5.2: Code expansion. Measured added or modified SLOCs. The original C version uses 30, 50 and 45 SLOCs for the *eval*, *grad* and *hesse* routine, respectively.

implementations. One possibility to reduce the effort needed to port an application from the CPU to the GPU are directive-based approaches such as PGI Accelerator [26] or OpenACC [27], which (similarly to OpenMP) allow incremental parallelization.

Even though the SLOCs give a good initial overview of the effort involved in writing the routines, I spent most of the time tuning the kernels for the respective architectures. Moreover, it felt easier to design the algorithms for the CPU than for the GPU because the CPU implementations required less optimization techniques than the GPU implementations (i.e. the programmer has to take care of: registers usage, local memory usage, coalescing requirements, high thread-level parallelism, ...).

Taking the learning-effort and the time spent to design and implement the algorithms into account, I rank the productivity of the programming paradigms with respect to the routines used in this thesis as follows:

1. OpenMP (CPU)
2. Intel intrinsics AVX (CPU)[1]
3. NVIDIA CUDA/ OpenCL (GPU)
4. Intel OpenCL (CPU)

As mentioned in Section 3.5 and outlined in Table 3.5, the NVIDIA OpenCL versions resulted in poor performance on the CPU and would have required many changes, which made is necessary to rewrite the NVIDIA OpenCL versions for the CPU. Moreover, using Intel OpenCL for the CPU felt counterintuitive.

Despite the increased effort for the GPU implementations compared to the CPU implementations and their similar performance results with respect to the results of this study, there are certain areas where GPUs achieve substantial speedups over their CPU

[1] please note that I only implemented the *eval* routine with intrinsics AVX. Hence, this rank might have changed if I had implemented more complex routines with intrinsics AVX. My decision to rank it second, is based on the fact, that it required only one afternoon to learn the necessary AVX intrinsics and to implement the intrinsics AVX version based on the existing OpenMP version.

counterparts [25], [21]. Therefore, GPUs can be of interest, if either maximum performance is desired or one considers productivity as a ratio of speedup over programming effort. Hence, the decision whether to use the CPU or the GPU very much depends on the underlying problem.

Moreover, an additional naive GPU implementation of Kernel *gpuEvalDense* (see Appendix A.1) only required little effort and still yields good performance results (see Appendix A.2). Hence, the design of GPU implementations with *good*-performance results does not necessarily require more time than the design of CPU implementations. However, if *peak* performance is desired, additional effort is required in order to pay attention to the many architectural features (e.g. coalescing, local memory, bank conflicts). The troublesome process of designing such an enhanced algorithm is outlined in Appendix A.2.

As far as *tool and library support* is concerned, the GPU-based paradigms start to catch up with the versatile support for CPUs. OpenCL and CUDA (for GPUs) offer GUI-based debuggers, profilers and an increasing number of libraries (e.g. cuFFT, cuBLAS, MAGMA[1], ViennaCL[2], AMD Accelerated Parallel Processing Math Libraries (APPML)).

[1] work in progress. CUDA implementation of some LAPACK routines
[2] OpenCL GPU linear algebra library

6. Conclusion

In the course of this thesis, I have evaluated the performance and productivity of modern vector processors on the example of a real-world MEG application. With respect to this application, the CPU and the GPU yield comparable performance both for single and double precision. Hence, I recommend OpenMP if productivity - as a ratio of speedup over programming effort - is the main goal and NVIDIA's CUDA or OpenCL APIs if maximum performance (for large problem sizes) is desired.

As it is evident from the results of this study, the key to achieve peak performance is a fundamental knowledge of the underlying CPU/GPU architecture and its optimization techniques. Looking at productivity, I consider the development of high-performance GPU applications more difficult than the development of high-performance CPU applications, since GPU implementations are sensitive to many parameters such as work-group size, coalescing requirements, shared-resource usage and many more. Nonetheless, high-performance CPU implementations require further attention as well, especially if the application should benefit for the SIMD capabilities of modern CPUs. However, the performance results demonstrate that despite the additional endeavours to utilize the increased SIMD capabilities of the used CPU, the routines examined in this study did not notably improve their performance due to vectorization.

Moreover, the question on whether to use the CPU or the GPU remains open. As this decision very much depends on the problem at hand. One solution to this question could be to analyze the problem in terms of available parallelism (i.e. does the problem exhibits enough parallelism to exploit massively parallel architectures like the GPU?) or its *compute to memory access ratio* (i.e. will the algorithm be memory bound?) and compare it to similar applications for which their CPU and GPU performance is known [21].

Furthermore, improved auto-vectorization and auto-parallelization capabilities of future CPU compilers could further ease the utilization of the increasing number of cores and the increasing vector-width of future CPUs. On the other hand, directive-based approaches like OpenACC and PGI Accelerator could decrease the effort involved in writing high-performance GPU applications.

I think that it is likely that the trend towards massively parallel architectures continues. For example, Intel's *Many Integrated Core* (MIC) architecture will have more than 50 cores and a SIMD-width of 512-bit. Furthermore, Intel's next CPU architecture called *Haswell* will increase its vector capabilities even further and support the AVX 2. GPUs tend to further increase their number of cores. While AMD/ATI already offers GPUs with more than 2000 cores per GPU (e.g. *Radeon HD 7870*, based on AMD's *Graphics-Core-Next* architecture), NVIDIA just introduced the *GeForce GTX 680* which is based on NVIDIA's new GPU architecture called *Kepler* and increases its core-count to 1536.

In the future, I plan to investigate the performance of AMD's *Graphics-Core-Next* architecture and NVIDIA's new *Kepler* architecture.

A. Further GPU Versions

This section outlines two additional GPU versions of Kernel *gpuEvalDense*. A naive version requiring only little effort is outlined in Section A.1, while Section A.2 illustrates a more laborious approach.

A.1. Naive Eval Routine

This version is split into two kernels, one sparse kernel (exactly the same as kernel *gpuEvalSparse*) and a dense kernel (*gpuEvalDenseNaive*). The control flow and workload distribution of this kernel is outlined in Figure A.1.

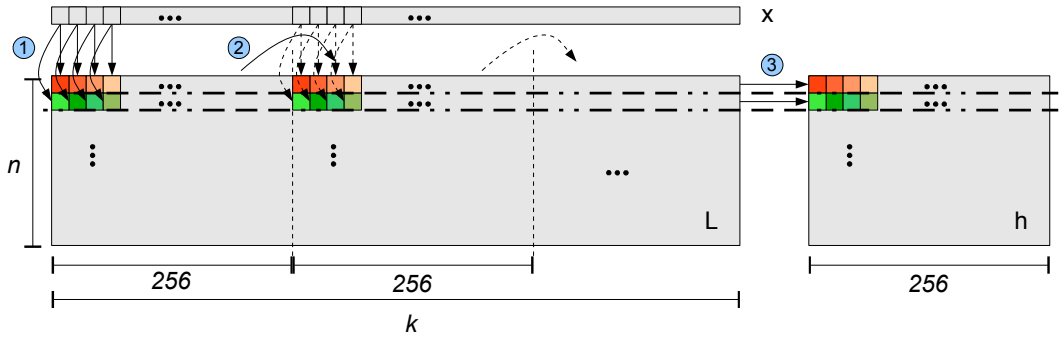


Figure A.1: Exemplary control flow and work distribution of two work-groups of kernel *gpuEvalDenseNaive*. The orange colors denote the responsibilities of different work-items within the same work-group (green analogous).

The control flow of each work-group is as follows:

- *Step 1*: Multiply the current chunk of the row with the current chunk of x.
- *Step 2*: Proceed with the next chunk until the end of the row is reached.
- *Step 3*: Store the private result[1] of each work-item to the auxiliary array h.

The auxiliary array h is then transferred to the host and reduced on the CPU-side.

Even this naive implementation requires knowledge about the underlying GPU architecture, as it pays attention to (1) the coalescing requirements and (2) the maximum utilization of the CUs. As depicted in Figure A.1 all work-items within a warp access consecutive data elements (1) of the matrix L and the vector x, which enables the architecture to coalesce these memory accesses. Furthermore, the work-group size is chosen to be 256, which allows for maximum utilization of the CUs (2). More precisely, if the usage of shared resources of a kernel does not limit the amount of parallel executing work-items (which is the case for this kernel), Fermi limits the number of concurrently running work-groups, warps and work-items to 8, 48 and 1536, respectively [18, p.137].

[1] residing in registers

Hence, $\lfloor 1536/256 \rfloor = 6$ work-groups suffice to saturate a CU, while 128 work-items per work-group would require $\lfloor 1536/128 \rfloor = 12$ work-groups per CU which is not possible due to the aforementioned limitations of the architecture. Performance analysis showed, that 256 work-items per work-group yield roughly 40% performance increase over the same kernel running only 128 work-items per work-group (compare: theoretical speedup: $(8 \cdot 128)/(6 \cdot 256) = 1.5$).

This kernel is very simple and provides a good performance (see Table A.1). However, it does not reuse the x vector for consecutive rows[1]. Kernel *gpuEvalDense* (see Section 4.4) and Kernel *gpuEvalDenseImproved* (see Appendix A.2) try to overcome this problem at the cost of additional local-memory usage and reduction operations.

A.2. Kernel *gpuEvalDenseImproved*

Despite of the good performance results shown in this section this version is not used by other routines, since it was originally considered as future work and has been implemented at the very end of my study.

Kernel *gpuEvalDense* (see Figure 4.3) suffers from two main disadvantages.

First, it only uses 32 work-items per work-group (i.e. one warp). Hence, only 8 warps can reside on one CU at a time. This is not enough to hide the memory latency of global memory [18, p.79]. Moreover, the extensive use of local memory would limit the number of residing warps on a CU to 11 for single precision and 5 for double precision (refer to Section 4.5), hence it is desirable to reduce the usage of local memory per warp but at the same time do not violate the coalescing requirements.

Second, work-groups of Kernel *gpuEvalDense*, processing the same columns of the matrix L, load x redundantly.

These two disadvantages are eliminated by the improved Kernel *gpuEvalDenseImproved* (see Figure A.2). Kernel *gpuEvalDenseImproved* reduces the use of shared memory per warp, while at the same time it increases the number of work-items per work-group from 32 to $nRows \times 32$, $nRows \in \{4,8\}$, so that each work-group consists of at least 4 warps. This allows to have more warps per CU than the original kernel, so that memory latencies can be hidden more efficiently. Additionally, divergence is restricted to work-items belonging to different warps (i.e. no performance penalty).

Additionally each work-group works on whole columns, such that x is not redundantly loaded.

The steps of Kernel *gpuEvalDenseImproved* are as follows:

- *Step 1*: One warp loads a portion of x to local memory[2].
- *Step 2 + 3*: Each warp processes its chunk of the current row (i.e. *Workload* (= 16) many chunks of the same row).
- *Step 4*: Each warp reduces the elements of its chunk to a single value and stores it to $h_{reduced}$.

[1] except for some cached elements in Fermi's L1 and L2 caches

[2] the exact size of the portion of x requires further evaluation, because it directly affects the used local memory and hence the available parallelism for each CU.

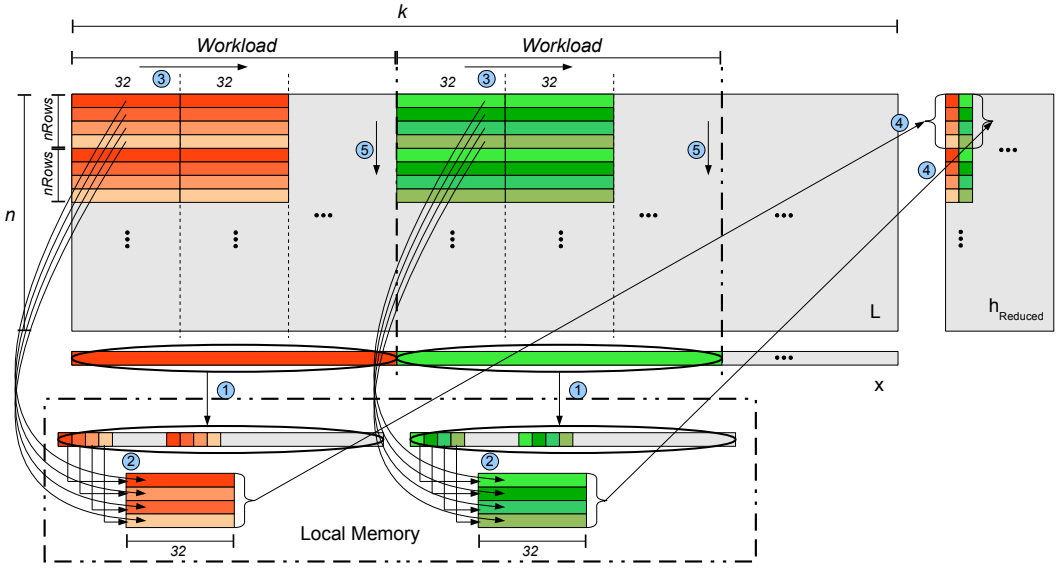


Figure A.2: Exemplary control flow and work distribution of two work-groups of Kernel *gpuEvalDenseImproved*. *Workload* denotes the number of blocks, each warp has to process. The orange colors denote the responsibilities of different warps within the same work-group (green analogous).

- *Step 5*: Each warp proceeds to the next row and continues with step 1 until all rows have been processed.

The performance of this improved version is compared to the other two *eval* routine versions in Table A.1.

	Single			Double		
	Naive	Original	Improved	Naive	Original	Improved
Runtime	4.8	4.8	4.2	11.3	10.6	7.7

Table A.1: Runtime (in ms) of three different versions of the *eval* routine using CUDA. *Naive*, *Original* and *Improved* denote the *eval* routine using Kernel *gpuEvalDenseNaive*, *gpuEvalDense* or *gpuEvalDenseImproved*, respectively.

Table A.1 shows, that the *eval* routines using Kernel *gpuEvalDenseNaive* and *gpuEvalDense* yield roughly the same performance, Kernel *gpuEvalDenseImproved*, on the other hand, outperforms the original implementation by 14% and 38% for single and double precision, respectively.

References

[1] R. Bordawekar, U. Bondhugula, and R. Rao. Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU. *Research Report RC25033, IBM TJ Watson Research Center*, 2010.

[2] H. M. Bücker, R. Beucker, and C. H. Bischof. Using automatic differentiation for the minimal p-norm solution of the biomagnetic inverse problem. In A. W. Heemink, L. Dekker, H. de Swaan Arons, I. Smit, and T. L. van Stijn, editors, *Shaping Future with Simulation, Proceedings of the 4th International Eurosim 2001 Congress, Delft, The Netherlands, June 26–29, 2001*. Dutch Benelux Simulation Society, 2001.

[3] H.M. Bücker, R. Beucker, and A. Rupp. The NINA Software Package: Software for the Solution of Neuromagnetic Inverse Large-scale Problems. Preliminary Manual, 2011.

[4] H.M. Bücker, R. Beucker, and A. Rupp. Parallel Minimum p-Norm Solution of the Neuromagnetic Inverse Problem for Realistic Signals Using Exact Hessian-Vector Products. *SIAM Journal on Scientific Computing*, 30(6):2905–2921, 2008.

[5] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*, volume 10. The MIT Press, 2007.

[6] A.R. Conn, N.I.M. Gould, and P.L. Toint. *Trust-region methods*, volume 1. Society for Industrial Mathematics, 2000.

[7] Khronos Group. The OpenCL Spezification. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, March 2012. Version 1.1.

[8] M. Hämäläinen, R. Hari, R.J. Ilmoniemi, J. Knuutila, and O.V. Lounasmaa. Magnetoencephalography—Theory, Instrumentation, and Applications to Noninvasive Studies of the Working Human Brain. *Reviews of modern Physics*, 65(2):413, 1993.

[9] Intel. A Guide to Auto-Vectorization with Intel C++ Compilers. http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/, October 2011.

[10] Intel. Introduction to Intel Advanced Vector Extensions. http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions/, October 2011.

[11] Intel. Using AVX Without Writing AVX Code. http://software.intel.com/en-us/articles/using-avx-without-writing-avx-code/, October 2011.

[12] Intel. Intel OpenCl SDK User's Guide. http://software.intel.com/file/39188, February 2012.

[13] Intel. Writing Optimal OpenCL Code with Intel OpenCL SDK. http://software.intel.com/file/39189, February 2012.

[14] D.B. Kirk and W.H. Wen-mei. *Programming Massively Parallel Processors: A Hands-on approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2010.

[15] A. Munshi, B. Gaster, T.G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 2011.

[16] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. *NVIDIA Whitepaper*, 2009.

[17] NVIDIA. CUDA C Best Practices Guide, March 2012. Version 4.0.

[18] NVIDIA. CUDA C Programming Guide, March 2012. Version 4.0.

[19] OpenMP. OpenMP Application Program Interface. http://www.openmp.org/mp-documents/OpenMP3.1.pdf, March 2012. Version 3.1.

[20] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82. ACM, 2008.

[21] P. Springer. Seminararbeit - Berkeley Dwarfs on CUDA. 2011. RWTH Aachen University. HPC-Group.

[22] P. Springer. Vectorization of Kegelspan's Intersection Method. 2011. RWTH Aachen University. HPC-Group. Internal documentation.

[23] C. Terboven, D. Schmidl, H. Jin, T. Reichstein, et al. Data and Thread Affinity in OpenMP Programs. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A solved Problem?*, pages 377–384. ACM, 2008.

[24] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010.

[25] V. Volkov and J.W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.*, pages 1–11. IEEE, 2008.

[26] S. Wienke, D. Plotnikov, D. an Mey, C. Bischof, A. Hardjosuwito, C. Gorgels, and C. Brecher. Simulation of Bevel Gear Cutting with GPGPUs—Performance and Productivity. *Computer Science-Research and Development*, pages 1–10, 2011.

[27] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC - First Experiences with Real-World Applications. 2012. Submitted for publication.