TECHNISCHE UNIVERSITÄT DRESDEN

CENTER FOR INFORMATION SERVICES
& HIGH PERFORMANCE COMPUTING
PROF. DR. WOLFGANG E. NAGEL

Proseminar "Rechnerarchitektur"

Programming of Heterogeneous Computer Systems using OpenACC

Adam Kalisz
(Mat.-No.: 3805612)

Professor: Prof. Dr. Wolfgang E. Nagel
Tutor: Robert Dietrich, Dr. Hartmut Mix

Dresden, June 16th, 2014

Contents

1 Introduction

Since the times, when frequency scaling became unfeasible, because every circuit known must abide the following equation[1]:

$$P = C \cdot V^2 \cdot f$$

there is a tendency to add a higher number of less capable execution units instead of building fewer, more capable but also significantly more power hungry execution units. This is, where massive parallelism comes into play. Graphics Processing Units once applicable to graphics tasks only, are now capable to compute general purpose calculations. The reason for it consists in fact, that the development of their particular small cores follows Moore's law too. [30] GPUs with this kind of capability became known as General Purposse GPUs. With the hardware in place, several approaches to programming for these GPGPUs or, synonymously and more generally, for accelerators, became quite commonplace. This work deals with an approach called compiler directives programming, in particular, with OpenACC and OpenMP 4.0. Mentions of as well as traditional multi-core CPU programming will be done for comparison. The latter serves as a baseline in measurements.

2 OpenACC and other Models for Heterogeneous Computer Systems Programming

The scope of OpenACC application is only on, or below a node level. A node is, somewhat simplified, one motherboard in a cluster. That means, with OpenACC only, there is no inter-node communication possible. Also, OpenACC on its own does not enable parallel programming of CPUs. For this, OpenMP can be used. It enables "host-directed execution with an attached accelerator device." [41, 1.2 Execution Model]

2.1 Programming Model

The OpenACC™ specification describes "the compiler directives, library routines and environment variables that collectively define the OpenACC™ Application Programming Interface (OpenACC API) for offloading programs written in C, C++ and Fortran programs from a *host* CPU to an attached *accelerator* device." [41] This is done through the use of *pragmas* in C/C++ or *guided comments* in Fortran and extends the ISO/ANSI standard. The ultimate goal is to enable portability across operating systems, various CPU and accelerator combinations and to provide an abstraction level that takes care of managing "data or program transfers between the host and accelerator," [41] accelerator startup and shutdown.
The programming model still allows the programmer to "augment information available to the compiler [and explicitly define] data local to an accelerator." [41, 1. Inroduction] Generally, OpenACC does not prevent hand-optimization of code for better performance. On the other hand, such optimization will most likely not meet the performance targets of thoroughly optimized OpenCL or CUDA written code for a particular device.

2.2 Execution Model

The execution model is host-directed and compute intesive regions of user application are offloaded onto the attached accelerator device, such as a GPU. [41, 1.2 Execution Model] There are typically two types of regions, the *parallel* and the *kernels* region. These are offloaded to the accelerator device and are executed there. While a *parallel* region contains work sharing loops, the *kernel* region contains one or more loops, which are executed as kernels on the accelerator. [41, 1.2 Execution Model] "Even in accelerator-targeted regions, the host may orchestrate the execution by allocating memory on the accelerator device, initiating data transfer, sending the code to the accelerator, passing arguments to the compute region, queuing the device code, waiting for completion, transferring results back to the host,

[1] P stands for power, C for capacitance, V for voltage and f for frequency

and deallocating memory. In most cases, the host can queue a sequence of operations to be executed on the device, one after the other." [41, 1.2 Execution Model] Typically, accelerator devices support two or three levels of parallelism, a coarse-grain, a fine-grain and a vector or SIMD parallelism. The coarse-grain parallelism aims to divide work between execution units. An example is one execution unit of a Streaming Multiprocessor (SM). Such design is employed in Fermi architecture based NVIDIA GPUs and accelerators. The fine-grain parallelism is employed inside an execution unit and is mostly implemented in form of threads. Finally, the vector or SIMD parallelism assign work to the most basic execution units such as a CUDA core or a shader. These levels of parallelism are implemented as *gang*, *worker* and *vector* in OpenACC, corresponding to coarse-grained, fine-grained and vector parallelism. Also, new in OpenACC 2.0, the hierarchy structure has following features: a *vector* must be contained inside a *worker* and a *worker* inside a *gang*, this is not optional anymore. [41, 1.7 Changes from Version 1.0 to 2.0] This implies a gang must be started first, when launching a compute region on the accelerator device. First, the device starts in *gang-redundant* mode (GR mode). In that mode, one vector lane in one worker in each gang executes the same code redundantly. If a loop or loop nest marked for gang-level work-sharing is reached, the program starts to execute in *gang-partitioned* mode (GP mode). There, the iterations of the loop are partitioned across gangs, but still with only one vector lane per worker per gang active. In either GR or GP mode, if only one worked or vector lane is active, the program is in *worker-single* (WS mode) or *vector-single* (VS mode) accordingly. Analogously to gang-partitioned mode, the program enters *worker-partitioned* mode (WP mode), if a loop or loop nest marked for worker-level work-sharing is reached. This activates all workers in a gang and mapps iterations of the loop onto them. If the loop is marked for both GP and WP, then the iterations of the loop are spread across all the workers of all the gangs. Analogously, should a *vector-partitioned* mode (VP mode) exist, iterations of the loop(s) will be partitioned across the vector lanes using vector or SIMD operations. A loop may be marked for one, two or three levels of parallelism, but the hierarchy has to be taken into account. The execution is started on the host in a single thread identified by a program counter and its stack, where it can be parallelized by using e.g. OpenMP API. The host thread, or one of the host threads, starts a parallel context on the device. A thread on the accelerator is a single vector lane of a single worker and a single gang. [41]

The implementation of barrier synchronization or locks across gangs, workers or vectors is not recommended. The execution model allows for an implementation, that executes some gangs before others. Barriers would, on the other hand, hurt portability, as not all gangs must be active at the same time. This applies similarly to workers and vectors as well. On some devices, parallel kernels may be started. Then, OpenACC directives may be executed by the host or accelerator thread. In the specification, the terms *local thread* and *local memory* are used to describe the thread, that executes the directive, or the memory associated with that thread independently of whether the thread executes on the host or the accelerator.

Most accelerators can operate asynchronosly with respect to the host thread. In such case, there is one or more activity queues. After enqueuing the operation, such as data transfers or procedure execution, the host thread can continue execution, while the device executes operations independently. The host may wait for all operations of a queue to complete. That still does not disable the execution of different activity queues, which may complete in any order. [41, 1.2 Execution Model]

2.3 Memory Model

Unlike a host-only program, host+accelerator program can have two or more separate memories. This is a big difference between OpenMP prior to OpenMP 4.0 and OpenACC. OpenMP was designed for shared memory systems and simultaneous multi-threading. OpenACC by design violates the first design point of OpenMP, which until recently did not allow for separate memory. In OpenACC, all data movement is directed by the host thread, typically using direct memory access (DMA) transfers. It is discouraged to assume that the device can read or write host memory. This may be possible, but mostly with a performance penalty. [41, 1.3 Memory Model] There are limits for the execution owing to following memory model properties:

- The bus between host and accelerator device, based mostly on PCIe, is very slow compared to the access of host memory by host or accelerator memory by accelerator.
- The accelerator memory is, with current implementations, limited and may prohibit operations on very large amounts of data, if there isn't any possibility, how to split this data up.
- Pointer operations will be affected by the separation of host and device memories.

Devices may implement weak consistency memory model among different execution units or within execution unit. In the second case, memory coherency is guaranteed by a barrier. Caches, which may be implemented in software or hardware, are managed implicitly by the compiler with hints from the programmer. [19, Execution Model, pp.6] Data movement is implicit and managed by compiler. Data transfers can be orchestrated, mostly as optimization, by the use of data environment. "Data in this environment have an explicit lifetime, from when it is created or allocated until it is deleted." [41, 1.3 Memory Model] More about the data environment behaviour will be presented through examples in section. 2.5

2.4 Directives Format

To OpenMP users, OpenACC directives format will be familiar, see subsection 2.1 Every directive is introduced by a *pragma* in C/C++ such as *#pragma acc* or *guided comment* such as `!$acc` or `c$acc` or `*$acc` in Fortran. Then, directives and an optional list of clauses follow. The API call is closed by a new-line character.

```
#pragma acc directive-name [clause [[,] clause]...] new-line
```

Some common clauses for instance are *if(clause)* or *sync(handle)*.

2.5 Example in OpenACC and Equivalent in OpenMP

In general, programs that shall execute many independent simple operations are relativelly easy and popular to implement on accelerators. Obvious examples could be vector or matrix addition, multiplication or other basic operations and simple combinations of those. These tasks tend to be very compute intensive and are essential in applications such as modelling of moving particles or changes across many measurement points as the main use cases. A great example would be Point Jacobi Iterative method. [25] Following few notable examples in C shall illustrate, how to parallelize single-threaded, simple but widely used routines.

SAXPY in OpenACC

Single-precision A times X plus Y is a very simple vector scale and add operation. [35]

```
void saxpy(
        int n,
        float a,
        float *x,
        float *restrict y)
{
#pragma acc kernels
for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

As the reader can see, there are a scalar and two arrays, all of them of float type. n stands for the size of the array. The restrict type qualifier just tells the compiler that the value accessed by the pointer was modified and will not be accessed in the future. That is a very easy optimization, which can lead to significant performance gain. The array y will be overwritten. The *#pragma* line signifies an OpenACC API call. The directive *kernels* can be used with parallelism-unsafe code, meaning, with a code, wherein write operations to a variable could happen simultaneously. In this case, the compiler will try to parallelize the for loop as well as it can, as there is obviosly no parallel write operation to the same variable. In such case, we could just as well change the line to

```
#pragma acc parallel loop
```

This would cause explicit parallel execution of the loop. [25] If we want to make this example a bit more performant, we can manage the data by hand.

```
void saxpy(
        int n,
        float a,
        float *x,
        float *restrict y)
{
#pragma acc data copyin(x[0:n])
#pragma acc parallel copy(y[0:n])
#pragma acc kernels
for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

We want to make read-writable array y and a readable array x in its whole length (0 through n) available to the device, this should leave less space for error to the compiler and should also improve performance by a large margin, if we changed array y and executed the for loop once again.

SAXPY in OpenMP

Quickly, we can translate the OpenACC version to OpenMP. The *parallel loop* directive has a simple translation to OpenMP. In case of OpenMP, the following code example would run only on a CPU (with shared address space memory etc.) [51]

```
void saxpy(
        int n,
        float a,
        float *x,
        float *restrict y)
{
#pragma omp parallel for
for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

To make this example run on an accelerator device, we need to call additional APIs. [46, p.10]

```
void saxpy(
        int n,
        float a,
```

```
        float *x,
        float *restrict y)
{
#pragma omp target data map(to:x[0:n])
#pragma omp target device(0) map(tofrom:y[0:n])
#pragma omp parallel for
for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

This makes the program a little bit more complex. We say, we want to run this loop on a target device 0 and that we want to make read-writable array y and a readable array x in its whole length (0 through n) available to the device. The *device(0)* can also be set through environment variable OMP_DEFAULT_DEVICE=<int> or if only one device is present, omited entirely. This example is meant for Intel Xeon Phi "Knights Corner" coprocessor. Why this is so will be explained in subsection 2.10.

2.6 New Features in OpenACC 2.0

The OpenACC 2.0 specification is about two times the OpenACC 1.0 [40] in length. This gives a hint, there are some exciting new features and other additions.

Function calls within compute regions

OpenACC 1.0 compilers rely on inlining function and subroutine calls within compute regions. [36] In OpenACC 2.0 there is a new *routine* directive. This instructs the compiler to build an accelerated version of a given function or subroutine, that may be called inside a device region. Optimization clauses, for *gang*, *worker*, *vector* and *seq* (sequential) parallelism are still available.

Unstructured data regions

In OpenACC 1.0, data regions are always structured and end with the lexical scope of a function. In OpenACC 2.0 `#pragma acc enter data` and `#pragma acc exit data` define a data region begin and end, which may help handling certain situations better. For example, begining a data region inside a function and ending it inside another function would not be possible in OpenACC 1.0. [41, 2.6.2 *Data Regions and Data Lifetimes*]

Nested parallelism

In OpenACC 2.0, it is now possible to include `#pragma acc parallel loop` or `#pragma acc kernels` regions inside any of them. This, combined with the clarification of allowable loop nesting hierarchy, can help getting more dynamic parallelism or optimize better.

device_type Clause

This helps with writing device specific code, which may be better optimized, without using complicated `#ifndef` workarounds. The following example shows, how defining vector unit width for more devices at once is much simpler. [41, 2.7.8 *device_type clause*]

```
#pragma acc parallel loop \
            device_type(nvidia) vector_length(256) \
            device_type(radeon) vector_length(512) \
                                vector_length(64)
```

atomic Directive

The *atomic* directive serves a purpose of a lock. It prevents other threads from writing to a variable. This can come in handy, if you want to increment a counter from multiple threads for instance. [41, 2.10 Atomic Directive]

Tile Clause

In OpenACC 2.0, locality can now be expressed better using the *tile* clause. This clause makes each loop in a loop nest split into two loops, an outer set of *tile* loops and an inner set of *element* loops. The argument to the clause is a list of tile sizes. That helps with data reuse and through that with optimizing for performance e.g. some matrix operations. [41, 2.7.7 *tile clause*]

default(none) Clause on *parallel* and *kernels* Directive

"The *default(none)* clause is optional. It tells the compiler not to implicitly determine a data attribute for any variable, but to require that all variables or arrays used in the compute region [such as *parallel*] that do not have predetermined data attributes to explicitly appear in a *data* clause for the compute construct or for a data construct that lexically contains the *parallel* or *kernels* construct." [41, 2.5.12 *default(none) clause*]

New Concepts

These are some of the additions, that help with optimization [41, 1.7 Changes from Version 1.0 to 2.0], [41, 2.7.2–2.7.4 gang, worker, vector clause].

- *gang-redundant*
- *gang-partitioned*
- *worker-single*
- *worker-partitioned*
- *vector-single*
- *vector-partitioned*
- *thread*

New API Routines

These can help with the interoperability with CUDA and OpenCL code. For instance, there is now a way to map an existing device memory allocation from CUDA or OpenCL to a variable in OpenACC. This is possible using [41, 3.2.25 *acc_map_data*] and [41, 3.2.26 *acc_unmap_data*]. There are other, mostly memory management related, new API routines. For detailed information, please consult the specification and the compilers handbook.

2.7 New Features in OpenMP 4.0

OpenMP 4.0 marks a short departure from the original concepts of OpenMP in some specific and well defined scenarios such as using an accelerator. OpenMP 4.0 tries to cover not only GPUs, but also coprocessors such as Intel Xeon Phi, Field-Programmable Gate Arrays and Digital Signal Processors. [14] OpenMP 4.0 was influenced by OpenACC as some companies, such as Cray, influence both industry standards. Because OpenACC was released in 2011, the ideas became an inspiration for the current

release of OpenMP, which was extended to cover accelerators more in general because of the scope OpenMP always covered. This took two years longer compared to OpenACC 1.0 since OpenMP ABI have to take into account the legacy of well over a decade and the scope of OpenMP to stay backwards compatible.

SIMD Constructs

The *simd* construct can be applied to a loop, so multiple iterations of that loop can be executed at once using SIMD instructions and for that matter SIMD or vector execution units. The OpenACC equivalent to *simd* construct is more or less the *vector* directive. Use of *simd* construct places some restrictions on the structure of associated *for-loops*. These must have *canonical loop form*. [49, 2.6 Canonical Loop Form, 2.8 SIMD Constructs] The *collapse* clause may specify, how many loops are associated with the construct. The parameter must be a positive integer value. If no *collapse* clause is present, the only loop accociated is the loop following the construct. The *safelen* clause defines, how many iterations can be run concurrently without breaking a dependence. In practice, this is the maximum vector length. [15] There are two other interesting clauses, the *linear(list[:linear-step])*, which creates a relationship between the variable's value and the iteration number. The other one is *align(list[:alignment])*, which is a performance oriented optimization. It defines an alignment for the list elements. The default is alignment for the architecture.
Following is a scalar product example where *simd* construct can be used.

```
void sprod(float *a, float *b, int n){
        float sum = 0.0f;
        #pragma omp for simd reduction(+:sum)
        for(int k=0; k<n; k++)
                sum += a[k] * b[k];
        return sum;
}
```

Helper functions can be also used inside a loop and still use the *simd* construct. `#pragma omp declare simd` or a very similar line would be used in such instance.

target Constructs

The *target* construct transfers the control flow to the target device. The transfer of control is sequential and synchronous. (Asynchronous transfer can be created as a workaround using a parallel for-loop e.g.). The transfer clauses control direction of data flow and an array notation is used to describe array length. (SAXPY in OpenMP, map(tofrom:y[0:n] e.g.) [46]
The *target data* construct does not include a transfer of control, the transfer clauses still control direction of data flow and the data transfered with it is valid through the lifetime of the target data region. [15] That means, the environment is lexically scoped.
The *target update* construct is used to update the data already present on the device.
The *target device* construct can be used to specify the device used for offloading computation too. The device identification is a positive integer number.
There is also a *define target* construct can be used to move data to device. Here is an example:

```
#pragma omp declare target new-line
declarations-definition-sequence
#pragma omp end declare target new-line
```

[49, 2.9.4 *declare target* Directive]

teams Construct

This construct is there for multi-level parallel devices, such as GPUs. This translates approximately to OpenACCs *gangs*. Following clauses can be used:

- *num_teams(int)*
- *thread_limit(int)*
- *default(shared | none)*
- *private(list)*
- *firstprivate(list)*
- *shared(list)*
- *reduction(reduction-identifier : list)*

On an Intel Xeon Phi, the number of cores is 61, so we use *num_teams(61)*. Each core has 4 threads, so we use *thread_limit(4)*.

Affinity, Concept of Places

Todays systems are of Non Uniform Memory Access (NUMA). That means, the access time to memory can be different, depending on the location in the system.
In todays designs of CPUs and the structure of nodes, some threads are closer to each other, than to other threads. More precisely, the execution units are not spread evenly in a server. To accomodate smarter scaling, to run certain threads on one processor package could be a great example, there are three predefined places

- *threads* - one place per hyper-thread
- *cores* - one place per physical core
- *sockets* - one place per processor package

other custom places can be defined by the user. Further, there are affinity policies available

- *spread* - spread OpenMP threads evenly among places
- *close* - pack OpenMP threads near master thread
- *master* - collate OpenMP threads with master thread

There is a clause *proc_bind(master | close | spread)* [49, 2.5.2 Controlling OpenMP Thread Affinity] for parallel regions, which sets the affinity policy, the policy for assigning threads to places. There is also an execution environment routine *omp_get_proc_bind*, which returns the affinity policy. This can be used for a subsequent nested parallel region that does not specify a *proc_bind* clause. [49, 3.2.22 *omp_get_proc_bind*]

New Environment Variables

There are also new environment variables. *OMP_PLACES* can be used to define additional places. [49, 4.5 OMP_PLACES] *OMP_PROC_BIND* can be used to define affinity and has parameters *true* or *false* or a comma separated list of *master*, *close*, *spread* such as:

```
setenv OMP_PROC_BIND false
setenv OMP_PROC_BIND "spread, spread, close"
```

The environment variable *OMP_DEFAULT_DEVICE* can be used to specify the default accelerator, which is a positive integer value. [49, 4.13 *OMP_DEFAULT_DEVICE*]

Other Notable Changes

These are for instance the *atomic* construct, which is in nature similar to the definition in OpenACC and the *task* construct, which has now extensions to support deep task synchronization for example. Support for Fortran 2003 is now also present. [48]

2.8 CUDA

Compute Unified Device Architecture (CUDA) [34] is a proprietary platform of NVIDIA. It encompasses hardware, optimized libraries, compiler direktives und extensions for C, C++, Fortran. Many other programming languages, such as (Haskell, Java, Lua etc.) can be "compiled", (translated may be more accurate) into CUDA or there are APIs available, to call CUDA kernels from those. The current version is CUDA 6. [32]

2.9 OpenCL

Open Computing Language (OpenCL) is a framework which builds on C99 and can be used for programming of CPUs, GPUs, DSPs, FPGAs. It is an open standard managed by the Khronos Group and has some support in most devices currently on the market though the breadth of support varies greatly and is driver and architecture dependent. The current version is OpenCL 2.0. [47]

2.10 Supported Platforms and Compilers

To the best of my knowledge, the following table is a summary and a short description of OpenACC and OpenMP support in compilers available at the time of writing. As most compilers do not cover a specification fully, or have some specialties, there are coments in parentheses. Interesting further information is linked. A question mark signifies that the information is not readilly available or is very hard to find. "Out of Tree" (OoT) means, there is a work in progress, but the feature set is not part of any official release, which may imply performance, feature set or reliability limitations of those implementations.

Compiler	**OpenACC**	**OpenMP 4.0** [42]	**Links**
GCC 4.8	No	No (OMP 3.1)	
GCC 4.9	OoT	Yes (*target* execution on host)	[44], [43]
LLVM/ Clang 3.4	No	OoT (OMP 4.0 targeted at v3.5)	[11]
IBM XL compiler family	No	Partial (OMP 4.0)	[9], [10]
Oracle	No	No (OMP 3.1)	[6]
Intel C++/Fortran Studio XE	No	Yes (XE2013 SP1)	[12]
Absoft Pro Fortran	No	No (OMP 3.0)	[8]
NAG Compiler 5.3.1 Fortran	No	No (OMP 3.0)	[29]
PGI Compiler 14.6	Yes	No (OMP 3.1?)	[18]
CAPS Compilers 3.4.5	Yes	No (N/A?)	[16]
Cray CCE 8.2	Yes	No (OMP 3.1)	[24]
Lahey/Fujitsu	No	No (OMP 2.0)	[27]
ROSE Compiler Framework	No	No (OMP 3.0)	[39]
HP C/C++	No	No (OMP 2.5)	[20]
Microsoft VC++	No	No (OMP 2.0)	[13]
PathScale EKOPath CS	No	No (OMP 2.5)	[23]
Open64	No	No (last release 2011, < OMP 2.5?)	[7]
OpenUH	Yes (?)	No (OMP 3.0, based on Open64)	[38], [22]
accULL (Research Compiler)	Yes (OACC 1.0)	No (last release end of 2013)	[21]

Some compilers allow for intermediate compilation of OpenACC accelerated code into OpenCL but they do not allow to generate the assembly directly. In this manner, OpenACC code can be run on Intel Xeon Phi without the need for the Intel compiler to support OpenACC.

If you posses a NVIDIA GPU (Quadro, Tesla, GeForce), you can use CUDA on Windows, Linux and Mac OS X. This platform is well supported and widely in use. There is plenty of material and developer support is strong. The code is not very portable, though there is progress in that area and there are ways to translate CUDA into OpenCL.
OpenCL is used in some extent by all major hardware companies. Intel and AMD are the leaders and both have plans to support OpenCL 2.0 broadly, though the current support is at OpenCL 1.2 which is the last version before OpenCL 2.0. NVIDIA supports mostly OpenCL 1.1 with plans for OpenCL 1.2 support through driver update it seems. The OpenCL driver support situation on anything besides Windows platform is not very strong and the quality varies greatly. Other industry players, such as ARM, Texas Instruments, Qualcomm, Samsung and others also provide some OpenCL support, but these are not strong players in the HPC area.

3 Example implementation

3.1 Motivation for Accelerated Sorting Algorithm

The main idea behind trying to sort on the GPU was the immense parallelism compared to a CPU. This could be used to sort e.g. "big data" [31] on a GPU, where they have been processed before. Also, sorting is not covered in most parallelization examples elsewhere, so this work was a short study in an, to the best of my knowledge, unexplored area[2] and an attempt to assess the viability of the implementation of the sorting algorithm "Bitonic Sort" [28] presented in this work as well.

3.2 Hardware and Software Environment

The environment for measurement was split between two accelerated systems and a CPU only system, which was private and therefore accessible at any time. It would be a valid comparison point to a certain point, since it is a dual socket system and the inter-processor communication can therefore be taken into account, which would not be possible in a single socket system.
This system probably has about the same performance as a brand new non quad-core laptop, but dual-core with hyper-threading and both memory channels allocated with faster DDR3 RAM. Measurement was not done on the system equiped with 2x Intel Xeon E5140, since this systems performance is not comparable to current HPC grade systems. This system was used solely as a base target for development, since some behaviour patterns are similar to newer systems.
Development was done on a laptop system with Intel Core i7-2620M, 4GB RAM, no accelerator device, Linux kernel 3.14 and GCC 4.9. Further adjustments in source code have been done on the measurement systems between test runs as needed and the system mentioned above since using all threads at 100% on a laptop is not feasible, if other work has to be done.
The kernel version has been obtained using `uname -r` Linux command.

Intel MIC [3]	**NVIDIA Tesla** [2]	**CPU only**
2xIntel Xeon X5680@3.33GHz	2xIntel Xeon E5620@2.40GHz	2xIntel Xeon E5140@2.33GHz
max. 24 (2x 6x 2) threads	max. 16 (2x 4x 2) threads	max. 4 (2x 2) threads
24GB RAM	24 GB RAM	20 GB (4x1 + 4x4) RAM
Intel Xeon Phi KC class	1xS2050 Fermi (12 GB RAM)	N/A
2.6.32.12-0.7-default	2.6.32.59-0.7-default	3.13-0.bpo.1-amd64
Intel C Compiler v14.0.2.144	PGI C Compiler 14.1 and 14.6	GCC 4.7.2

The PGI C Compiler 14.6 was only used for testing. For performance measurement, PGI C Compiler 14.1 was used.
Since the effectiveness of the execution drops rapidly[3], if we want to execute the algorithm on more threads, than actually available, the CPU only measurement will be limited to the maximum number of threads available on each system.

[2] At least for the autor
[3] The data to back up this claim can be found in appendix section 7

3.3 Bitonic Sort

Bitonic sort [26], [28], [45] has a complexity of $\Theta(n \cdot log(n)^2)$.
The input is an array of positive integers. We split this array into half resursively so that we at some point get an array with one element. Such array is by definition sorted. Then, we merge this array with another. This we do in parallel for many such one-element arrays. Each second array is sorted descreasingly. We get a small "zig-zag" type array, for better imagination. Then we compare the first element of the first array, with the first element of the second array and the second element of the first array with the second element of the second array etc. This gets repeated until we do not have any remaining array of the same size. Then, the array is sorted. Obviously, this simple implementation does not sort arrays, whose length cannot be described with 2^n.

3.4 Implementation

The actual implementation of bitonic sort using OpenACC in particular turned out to be difficult for various, partly unforeseable, reasons. The original code, which was implemented in a similar manner to the implementation by Hans Werner Lang at Fachhochschule Flensburg [28], performed worse when parallelised with OpenMP or OpenACC. A second implementation, based partly on [1] did not produce valid results beoynd 4 threads (using OpenMP), so an implementation using OpenACC was not feasible. The last and current implementation is based partly on the implementation by Prof. Dr.-Ing. Philipp Slusallek and Javor Kalojanov PhD at Saarland University [45], which was, as I have been informed by Dr. Kolojanov, abstracted from the CUDA SDK. This implementation was used for parallelization OpenACC and OpenMP, which will be coverd in the following subsections. The current code provided valid results for the OpenMP CPU only implementation and was fairly easy to implement. The implementation seemed to perform well so the next step, parallelization using OpenACC seemed feasible.

3.4.1 Parallelization using OpenACC

After getting to know the PGI C Compiler used on the OpenACC equiped system, where the C flags are different compared to GNU C Compiler and Intel C/C++ Compiler and getting to know the flags for OpenACC (-acc,-Minfo=accel,time e.g.), the testing could begin. After a fair amount of work and research, two important findings came out. First, with minor changes to the *pragma* in the implementation, either, the code performed in single thread and produced validated results, or the code perfomed in parallel and did produce unsorted result for small sized arrays and increasingly better sorted results for larger arrays. The performance for single thread was not feasible for any further use, the performance for for the parallel implementation seemed very competitive for an array size of 268435456 2^44 compared to 12 thread CPU only (2xIntel Xeon E5620@2.40GHz) implementation, but the results still were not sorted entirely and therefore could not be used for comparison with verifiably sorted arrays. The likely reason for this behaviour is the different approach to the use of memory. Where OpenMP implies shared memory, OpenACC does not by default assume shared memory and pointers in a for loop are not executed atomically by consequence. With this finding, embedding *atomic* pragmas into the incriminating loop would most likely resolve the problem, and, as a side effect, have probably some negative performance impact. Implementing *atomic* pragmas turned out to be impossible with the current PGI C Compiler [18, Section 2.6.6. OpenACC Atomic Support] as only atomic addition and subtraction are supported as of PGI C Compiler 14.6; in 14.1, they are not supported at all. As of 14.4, there is a possibility of embedding CUDA code. There, the *atomicCAS* - compare and swap [33], might actually solve the issue but implementing that would be beyond the scope of this work. Because the code does not produce valid results, only the code for the parallel implementation will be provided with comments suggesting the use of atomics in the code. No specific performance results can be provided, because the comparison would not be useful for obvious reasons.

Parallelization using OpenMP

Parallelizing for the CPU only execution was easy, after the original code was written sufficiently well. At least for a few threads, the speedup was quite good as well. The story changes quickly, when speaking of the Intel Xeon Phi. There are two options for running the code on Intel Xeon Phi. There is the native option (use `-mmic` flag) and the possibility of offloading from host. The main implementation difference between the OpenACC and the OpenMP version is, the OpenACC loop nest is executed as a whole on the GPU. For the OpenMP, only the most inner loop is actually offloaded to the accelerator. The question, which of these approaches is more efficient remains to be answered. For reasons unknown at this point, the native version did not work. It might be some technicality, like missing libraries or wrong setup or there is a bug in the code, which exposes itself only on the Intel Xeon Phi, as the architecture of the cores is so different compared to regular, current day CPUs. The offloaded version did perform quite poorly, but not as bad as a single threaded OpenACC code on a GPU. Also, the array actually was verified to be sorted. This really has to do with the design differences between OpenACC and OpenMP. The actual implementation can be found in the appendix 7, some performance measurements will be provided in the subsection 4.2 since this is obviously a work in progress.

3.5 Limitations

As mentioned in subsection 3.3, bitonic sort as is cannot sort arrays, whose length cannot be described with 2^n. So we can sort an array of length $2 \cdot 2^n$, such as 8, but not 7 or 9 or, for that matter, any other odd number besides 1, which is axiomatically sorted. This matter is mentioned in subsection 5.2 as well. Further limitations continue to be the quality of the algorithm itself. There are some other more efficient algorithms available, such as RADIX sort in certain cases or parallel implementations of quick or merge sort, but finding the most efficient algorithm was not the objective of this work, it was to delve into programming in OpenACC and using the OpenMP for comparison. This does not mean, the performance comparison is out of place or that it is not a useful application, this is merely a hint there might be ways to optimize on the algorithm level.
Of course, another limitation of the OpenACC accelerated code lays in the lack any multi-platform free and open-source compiler that would support AMD manufactured accelerators as well. GCC targets NVIDIA products mainly. [50], [17], [37] The OpenMP code for Intel Xeon Phi would also be harder to run with GCC, than with the Intel compiler as the OpenMP 4.0 support still is not feature complete. The *target* directive executes on host. [44]
The limitation of the testing done are non consistent CPUs between systems, different Linux kernels used and different compilers used. The implementation for accelerators is not CPU bound because the main work is done on accelerator. Also the random number array is not measured, so it does not influence the runtime. What does matter is, that the array is different each time, so there is some variation between runs. This error can be eliminated using simple arithmetic mean on a statistically significant number of runs. Getting exact runtimes also was not the focus of this particular testing, it was the scaling and speedup of particular implementations.

4 Experimentation Results

4.1 Theoretical Targets

The main target is to achieve performance similar to mergesort without the need to use a helper array, since memory is scarse on accelerators at present. Also, the implementation should scale well and idealy sort any array size. See subsection 5.2

Comparison to Single-Threaded Mergesort

In theory, if every comparison in a step of bitonic sort can be executed in parallel, the complexity of bitonic sort should become $\Theta(n \cdot log(n))$. That is the complexity of mergesort as well. Mergesort is

a well known algorithm similar to bitonic sort in its approach of the problem. The main difference is, mergesort is suited best for sequential data. Also, naïve mergesort implementations need a helper array of the same size as the array to be sorted. [26] Mergesort is used widely because of its better worst case behaviour. Also, mergesort is very fast and using bitonic sort is very hard to beat. It the measurements included in this work, mergesort was the best for single-threaded scenarios across the board, but this advantage disapeared when using at least 2 cores for smaller array sizes and 4 cores for larger arrays. With larger array sizes, there would be a growing need for more cores to keep pace with single-threaded mergesort. The implementation of mergesort can be found in the appendix 7.

4.2 Results

Only a subset of measurements is included, since the focus is on scaling and behaviour, not so much on absolute numbers. Also, since the performance of offloading to Intel Xeon Phi is not comparable to the CPU only implementation, only a few measurements for this implementation have been made. See the graph in appendix 2. The axises are in logarithmic scale, so even a small difference is a big one. Actually, the runtime difference between the fastest run (70 seconds) of bitonic sort and the slowest (595 seconds) is about 525 seconds for the largest array, the Intel Xeon Phi implementation needs 4510 seconds to complete, which is made obvious by the fact, that the point is not included in the graph. We can also observe the overhead of offloading and the overhead for many OpenMP threads, both of which is substantial. For small arrays, using one or two threads may be the most efficient way to sort them.

4.3 Scaling

For the pure CPU version of bitonic sort, the scaling is near linear first and with more than 8 threads, the efficiency drops. Sometimes, more threads makes the computation slower. Interestingly, thread counts of power of two or counts matching the number of physical cores seems to improve the performance a bit. Some scaling still occurs with hyper-threads but the extent depends on hardware. In such a case, more information about affinity could boost the performance. The use of more threads than physically available is strongly discouraged since the performance drops rapidly; most likely because of context switching. See section 7

5 Discussion

5.1 Lessons learned

This work was a valuable lesson in many areas. The relativelly new specifications have been studied, programms using OpenACC and OpenMP were created. New approaches to problems had to be found constantly, which was a motivation for further research, a great deal of thinking and mostly lots of communication and managing. That enabled better understanding of the process of writing a scientific work and discovering new considerations when writing code such as dependence of iterations of a loop. Most of all, this work showed the importance of good time management as that was always a problem, when unexpected issues arised.

5.2 Future Development

There are plans to implement a combination of bitonic and merge sort or an algorithm that splits any array into subarrays sortable using bitonic sort, so that we can use the parallelism of accelerators, but at the same time sort any array. Because each number can be split in a sum of powers of 2 such as $7 = 2^2 + 2^1 + 2^0$, this aproach should work. The main obstacle is efficient allocation of these powers to threads and the memory efficient merging of sorted arrays.
More imediate plans are to make the OpenACC version of the code actually work. This should be possible using the CUDA workaround mentioned in subsubsection of 3.4.1 or the *atomic* directive, when ac-

tually fully implemented in the compiler used. There should be more work concerning the performance of both OpenACC and OpenMP implementations, which is not satisfactory as is. Better performance should be achievable through better understanding of the intricacies and by exploiting more debugging techniques. Also, the measurements could be done on another system using a NVIDIA Kepler architecture based accelerator. [5] Ideally, with better compiler support, measurement on AMD GCN [4] based accelerators could be done too. Last but not least, serious measurement should be done on a system with the same software and hardware equipment, such as the same Linux kernel, compiler, the same CPU(s), motherboard and RAM.

6 References

Please note, the online links have been obtained or verified to be available by 16th of June 2014. Throughout the work, the essential information is provided, most of the online information should be accessible in the same, or slightly changed form well beyond the date this work stays current. That means until the next major release of the OpenACC or OpenMP specification, related compilers and hardware.

References

[1] Byrne Students Summer Research 2012. Bitonic Sort: OpenMP Implementation. http://www.cs.rutgers.edu/~venugopa/parallel_summer2012/bitonic_openmp.html, 2012.

[2] Anandtech - Anand Lal Shimpi. NVIDIA's Fermi: Architected for Tesla, 3 Billion Transistors in 2010 - Architecting Fermi: More Than 2x GT200. http://www.anandtech.com/show/2849/3, September 2009. This an analysis of the NVIDIA Fermi architecture.

[3] Anandtech - Johan De Gelas. The Xeon Phi at work at TACC. http://www.anandtech.com/show/6451/the-xeon-phi-at-work-at-tacc, November 2012. This an analysis of the Intel Xeon Phi, Knights Corner class architecture.

[4] Anandtech - Ryan Smith. AMD's Graphics Core Next Preview: AMD's New GPU, Architected For Compute. http://www.anandtech.com/show/4455/amds-graphics-core-next-preview-amd-architects-for-compute/, December 2011. This an analysis of the AMD GCN architecture.

[5] Anandtech - Ryan Smith. NVIDIA GeForce GTX 680 Review: Retaking The Performance Crown - The Kepler Architecture: Fermi Distilled. http://www.anandtech.com/show/5699/nvidia-geforce-gtx-680-review/2, March 2012. This an analysis of the NVIDIA Kepler architecture.

[6] Oracle Corporation and/or its affiliates. Oracle Solaris Studio 12.3 Overview - OpenMP 3.1 for Parallel Programming. http://docs.oracle.com/cd/E24457_01/html/E21989/gjggm.html, January 2012. Compiler information library, OpenMP specifics, Compiler version 12.3.

[7] Computer Architecture and Parallel Systems Laboratory. Open64 Overview. http://www.open64.net/, 2014. Compiler website.

[8] ABSOFT Corporation. Absoft's Pro Fortran 2014 tool suite. http://www.absoft.com/Absoft_Linux_Compiler.htm, 2014. Compiler website.

[9] IBM Corporation. Advanced optimizing compiler for selected Linux distributions. http://www-03.ibm.com/software/products/en/xlcpp-linux, April 2014. Compiler website, Compiler version 13.1.

[10] IBM Corporation. XL Fortran compiler maximizes application performance on IBM Power Systems. http://www-03.ibm.com/software/products/en/xlfortran-linux, April 2014. Compiler website, Compiler version 15.1.

[11] Intel Corporation. An implementation of the OpenMP C/C++ language extensions in Clang/LLVM compiler. http://clang-omp.github.io/, June 2014. Github compiler branch webpage.

[12] Intel Corporation. Intel® Composer XE Suites - What's new. https://software.intel.com/en-us/intel-composer-xe#pid-3400-864, April 2014. Compiler website, most current version Intel® C++ Compiler XE 14.0.3.

[13] Microsoft Corporation. OpenMP in Visual C++ (Visual Studio 2013). http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx, 2014. MSDN Library.

[14] Dr. Michael Klemm and Christian Terboven. Die wichtigsten Neuerungen von OpenMP 4.0. http://www.heise.de/developer/artikel/Die-wichtigsten-Neuerungen-von-OpenMP-4-0-1915844.html, July 2013. This article is written in German.

[15] Dr. Michael Klemm, Intel. OpenMP 4.0 for SIMD and Affinity Features with Intel® Xeon® Processors and Intel® Xeon Phi™ Coprocessors. https://software.intel.com/en-us/articles/openmp-40-for-simd-and-affinity-features-with-intel-xeon-processors-and-intel-xeon-phi, October 2013. This is a series of videos.

[16] CAPS entreprise. CAPS Compilers 3.4.5. http://www.caps-entreprise.com/resources-and-support/latest-releases/, June 2014. Compiler releases website, the company went bankrupt in June 2014.

[17] Evgeny Gavrin, gcc@gcc.gnu.org. OpenACC branch. https://gcc.gnu.org/ml/gcc/2013-09/msg00235.html, September 2013. Mailing list entry.

[18] The Portland Group. PGI Release Notes - Version 14.6. https://www.pgroup.com/doc/pgirn.pdf, June 2014.

[19] Oscar Hernandez and Richard Graham. Introduction to OpenACC. https://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/IntroOpenACC.pdf, 2012. Application Performance Tools Group at Oak Ridge National Laboratory.

[20] L.P. Hewlett-Packard Development Company. Parallel Programming Using OpenMP. http://h21007.www2.hp.com/portal/download/files/unprot/aCxx/Online_Help/parallelprog.htm, 2010. Compiler website, OpenMP specifics.

[21] La Laguna University High Performance Computing Group. accULL release 0.3. http://accull.wordpress.com/2013/11/28/accull-release-0-3/, November 2013. Release notes.

[22] University of Houston High Performance Computing Tools group, Department of Computer Science. OpenUH - Open-source UH Compiler. http://web.cs.uh.edu/~openuh/, 2013. Compiler website.

[23] PathScale Inc. EKOPath™ Overview. http://www.pathscale.com/ekopath-compiler-suite, 2013. Compiler website, EKOPath 5.

[24] James Beyer, Cray Inc. S4708: Comparing OpenMP 4.0 Device Constructs to OpenACC 2.0. http://nvidia.fullviewmedia.com/gtc2014/S4708.html, March 2014. The OpenMP webpage states OMP 3.1 support in the current Cray Compiler Environment 8.2. In a talk at GTC14, mentions OpenMP 4.0 support. That video is no longer available and cannot be found elsewhere. Probably, OpenMP 4.0 support is a work in progress.

[25] Jeff Larkin, NVIDIA. Getting Started with OpenACC. http://www.youtube.com/watch?v=0e5TiwZd_wE&list=PL5B692fm6--uzIU4ccn9UQs61DQA22qDG&index=2, October 2013. This presentation is split in two videos.

[26] Donald Ervin Knuth. *The Art of Computer Programming*, volume 3, pages 159–168, 232–234. Addison Wesley, 1973. Mergesort and bitonicsort.

[27] Inc. Lahey Computer Systems. Lahey/Fujitsu Linux64 Fortran v8.1. http://www.lahey.com/linux64.htm#openmp, 2014. Compiler website.

[28] Prof. Dr. Hans Werner Lang. Sorting Networks - Bitonic Sort. http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm, March 1997. Updated on 18th of Mai 2010.

[29] The Numerical Algorithms Group Ltd. NAG Fortran Compiler. http://www.nag.com/nagware/np.asp, 2014. Compiler website.

[30] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 38(8), April 1965. http://www.computerhistory.org/semiconductor/assets/media/classic-papers-pdfs/Moore_1965_Article.pdf.

[31] NVIDIA Corporation. Big Data Analytics, Data Science & Machine Learning. http://www.nvidia.com/object/data-science-analytics-database.html, 2014. This should be used as an illustration, what were the motivations.

[32] NVIDIA Corporation. CUDA 6 Production Release. https://developer.nvidia.com/cuda-downloads, February 2014.

[33] NVIDIA Corporation. CUDA Toolkit Documentation v6.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomiccas, February 2014.

[34] NVIDIA Corporation. NVIDIA® CUDA™. http://www.nvidia.com/object/cuda_home_new.html, 2014.

[35] NVIDIA Corporation - Developer Zone. OpenACC: Directives for GPUs. http://devblogs.nvidia.com/parallelforall/openacc-directives-gpus/, March 2012.

[36] NVIDIA Corporation - Developer Zone. 7 Powerful New Features in OpenACC 2.0. https://devblogs.nvidia.com/parallelforall/7-powerful-new-features-openacc-2-0/, February 2014.

[37] Oak Ridge Leadership Computing Facility. OLCF Lends Expertise for Introducing GPU Accelerator Programming to Popular Linux GCC Compiler. https://www.olcf.ornl.gov/2013/11/14/olcf-lends-expertise-for-introducing-gpu-accelerator-programming-to-popular-linux-gcc-compiler/, November 2013.

[38] Xiaonan Tian (University of Houston). S4343 - OpenUH: An Open Source OpenACC Compiler. http://www.openacc.org/node/400, March 2014. Conference keynote.

[39] Community of the ROSE compiler framework developed at Lawrence Livermore National Laboratory. ROSE Compiler Framework/OpenMP Support. http://en.wikibooks.org/wiki/ROSE_Compiler_Framework/OpenMP_Support, January 2014. ROSE Compiler wikibook.

[40] OpenACC Corporation. The OpenACC™ Application Programming Interface Version 1.0. http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, November 2011.

[41] OpenACC Corporation. The OpenACC™ Application Programming Interface Version 2.0. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf, June 2013. Corrected in August, 2013.

[42] OpenMP Corporation. Openmp compilers. http://openmp.org/wp/openmp-compilers/, April 2013.

[43] GNU project and the GCC developers. GCC 4.9 Release Series - Changes, New Features, and Fixes. https://gcc.gnu.org/gcc-4.9/changes.html, June 2014. Release notes.

[44] GNU project and the GCC developers. GCC wiki - OpenMP. https://gcc.gnu.org/wiki/openmp, March 2014. GCC wiki page.

[45] Prof. Dr.-Ing. Philipp Slusallek and Javor Kalojanov PhD. Sorting in Parallel. https://graphics.cg.uni-saarland.de/fileadmin/cguds/courses/ws1213/pp_cuda/slides/06_-_Sorting_in_Parallel.pdf, 2012–2013. I was informed by Mr. Kolojanov, the implementation idea stems originally from NVIDIA CUDA Toolkit Documentation: http://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-sorting-networks.

[46] Christian Terboven. OpenMP 4.0. https://www.zki.de/fileadmin/zki/Arbeitskreise/SC/webdav/web-public/Vortraege/Darmstadt_2013/OpenMP_4_Overview.pdf, October 2013.

[47] The Khronos Group. The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/, March 2014.

[48] The OpenMP Architecture Review Board and OpenMP Corporation. OpenMP 4.0 API Released. http://openmp.org/wp/openmp-40-api-released/, July 2013.

[49] The OpenMP Architecture Review Board and OpenMP Corporation. OpenMP Application Program Interface Version 4.0. http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, July 2013.

[50] Thomas Schwinge, gcc-patches@gcc.gnu.org. Initial submission of OpenACC support integrated into OpenMP's lowering and expansion passes. https://gcc.gnu.org/ml/gcc-patches/2013-11/msg00623.html, November 2013. Mailing list entry.

[51] Tim Mattson, Intel. Introduction to OpenMP. https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG, December 2013. This is a series of videos.

7 Appendix

This data was measured on the second test system with 2x Intel Xeon E5620@2.40GHz and 16 physical threads. We can observe a significant drop in performance with 20 threads, which continues to be the

case till 32 threads, where we observe another significant drop. The most efficient execution seems to be the execution on physical cores only, without using hyper-threading. For small arrays, such as 2^{11} and 2^{12} in this instance 1, parallel execution actually can be slower, because there is thread communication and OpenMP environment overhead. For single threaded execution was therefore used a binary compiled without the corresponding OpenMP C flag.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define ASCENDING 1
#define DESCENDING 0

struct timespec startwtime, endwtime;
double seq_time;

__declspec(target(mic))
int *genArray;
__declspec(target(mic))
int arraySize;
int power = 0;

void generator(int size);
void printArray(int size);
char verify(int size);
void getPower(int size);

int main(int argc, char *argv[]){

    clock_gettime(CLOCK_REALTIME,&startwtime);
    srand((unsigned int)startwtime.tv_nsec);

    printf("The array size you want (integer): \n");
    scanf("%d", &arraySize);
    if(arraySize == 0){
        printf("Really, type in an integer larger 0...\n");
        scanf("%d", &arraySize);
    }

    genArray = malloc(arraySize*sizeof(int));
    generator(arraySize);

    clock_gettime(CLOCK_REALTIME,&startwtime);

    getPower(arraySize);

    if(power != 0){
    #pragma omp target data map(tofrom:genArray[0:arraySize])
    {
    #pragma acc data copy(genArray[0:arraySize])
for (int k = 2; k <= arraySize; k *= 2) //Parallel bitonic sort
    {
```

```
        for (int j = k / 2; j>0; j /= 2) //Bitonic merge
        {
            int cnt = arraySize;
            #pragma omp target
            #pragma omp parallel for
            #pragma acc kernels loop independent \
                present(genArray[0:arraySize])
            for (int threadid = 0; threadid < cnt; ++threadid)
            {
                //#pragma acc atomic capture --> speculation
                {
                // XOR
                int index = threadid ^ j;
                // Which section of the array are we in?
                if (index > threadid)
                {
                    if ((threadid & k) == 0) // ascending - descending
                    {
                        if (genArray[threadid] > genArray[index])
                        {
                            int t;
                            t = genArray[threadid];
                            genArray[threadid] = genArray[index];
                            genArray[index] = t;
                        }
                    }
                    else
                    {
                        if (genArray[threadid] < genArray[index])
                        {
                            int t;
                            t = genArray[threadid];
                            genArray[threadid] = genArray[index];
                            genArray[index] = t;
                        }
                    }
                }
                }
            }
        }
    }
}
}
}

    clock_gettime(CLOCK_REALTIME,&endwtime);
    seq_time = (double)((endwtime.tv_nsec - startwtime.tv_nsec)/1.0e9
                + endwtime.tv_sec - startwtime.tv_sec);

    if(verify(arraySize) == 1){

        printf("The array is verified to be sorted!\n"
```

```
                "It sorted in %f s\n", seq_time);
    } else{
        printf("Something went wrong, because the array isn't sorted!\n"
                "It still took %f s\n to complete", seq_time);
    }

    free(genArray);
    return 0;
}

// HELPER FUNCTIONS

/* Computes the power of 2, by which an array can be expressed */
void getPower(int size){
    int i;

    for(i=1; i<arraySize; i*=2){
        power += 1;
    }
}

/* Generates a semi-random array of integers */
void generator(int size){
    int i;

    for(i = 0; i < size; i++){
        genArray[i] = (int) rand() % 3571;
    }
}

/* Function for printing an array */
void printArray(int size){
    int i;
    for (i=0; i < size; i++){
        if(i % 10 == 0)
        printf("\n");
        printf("%d\t", genArray[i]);
    }
}

/* Function for verifying */
char verify(int size){
    int i;
    for(i = 1; i < size; i++){
        if(genArray[i - 1] <= genArray[i]){
        } else {return 0;}
    }
    return 1;
}
```

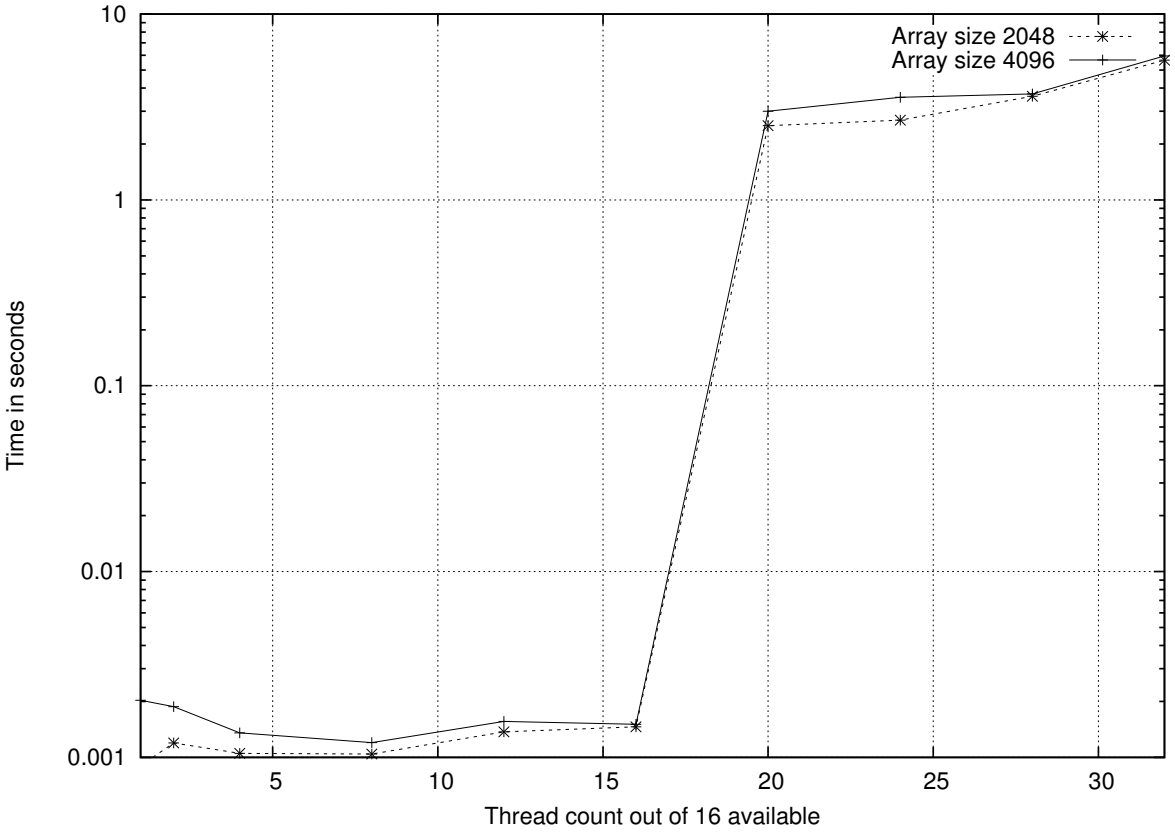

Figure 1: Scaling with the number of threads

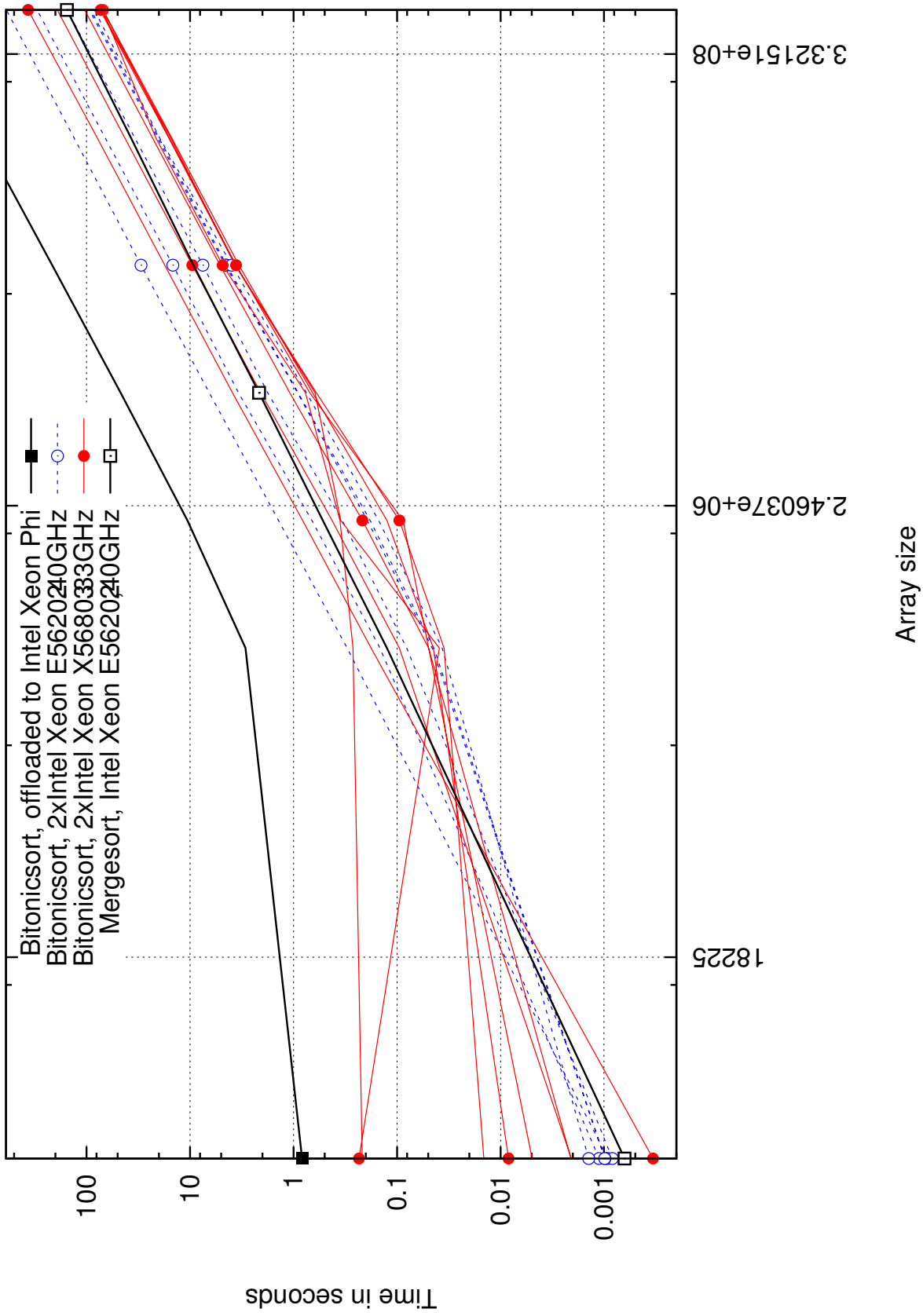

Figure 2: Measurement results, scaling behaviour observation

Acknowledgments

I would like to express appreciation to Prof. Dr. Wolfgang E. Nagel and Dr. Bernd Trenkler for creating an environment, in which it was pleasant to write this work and which was inspiring to me. I would further like to express deepest gratitude to my supervising tutors Robert Dietrich and Dr. Hartmut Mix for supporting and kindly giving me advice. Last, but not least, I would like to thank Daniel Molka and Alex Grudl for assistance with miscellaneous technical matters concerning testing.

Copyright Information