

# Student Cluster Competition 2016

## — Report —

Arbeitsbereich Wissenschaftliches Rechnen

Fachbereich Informatik

Fakultät für Mathematik, Informatik und Naturwissenschaften

Universität Hamburg

Vorgelegt von:

Sönke Behrendt,

Julian Frangopoulos,

Philip Gawehn,

Jesko Regenthal,

Michael Straßberger,

Kristina Tesch,

Thomas Walther,

Rasmus Warrelmann

Betreuer:

Dr. Michael Kuhn,

Anna Fuchs

Hamburg, den 30.11.2016

# Abstract

The International Supercomputing Conference (ISC <sup>1</sup>) is an annual global conference and exhibition for High Performance Computing. New and established companies in the High Performance Computing field are able to present their products and discuss the needs of their potential customers. Additionally, several workshops are held to communicate the knowledge gained in research groups. One major event is the announcement of the new TOP500 list of the 500 best performing supercomputers world-wide. Another event at the ISC is the Student Cluster Competition (SCC) where student teams from different universities from around the world come together and compete for the best performing cluster system. This report summarizes the experiences of the team from the Universität Hamburg at the SCC of ISC'16.

---

<sup>1</sup><http://www.isc-hpc.com/id-2016.html>

# Contents

<b>1</b>	<b>Introduction and Motivation</b>	<b>4</b>
1.1	Task and Rules . . . . .	5
1.2	Booth . . . . .	6
<b>2</b>	<b>System Configuration</b>	<b>7</b>
2.1	Hardware Overview . . . . .	7
2.2	Operating System . . . . .	9
2.3	Filesystem . . . . .	11
2.4	Spack Module System . . . . .	13
2.5	Installation . . . . .	14
<b>3</b>	<b>Benchmarks</b>	<b>19</b>
3.1	HPL . . . . .	19
3.2	HPCC . . . . .	23
<b>4</b>	<b>Applications</b>	<b>27</b>
4.1	WRF . . . . .	27
4.2	Splotch . . . . .	34
4.3	Graph500 . . . . .	44
4.4	CloverLeaf . . . . .	50

# 1 Introduction and Motivation

In July 2016, the 5th HPCAC-ISC Student Cluster Competition (SCC) took place at the ISC'16 Conference and Exhibition in Frankfurt (Main) Germany.



(a) ISC 2016<sup>1</sup>



(b) HPC Advisory Council<sup>2</sup>

This was the third time a team from the Universität Hamburg participated in the SCC. We all completed an introductory course in HPC at the German Climate Computing Centre (Deutsches Klimarechenzentrum, DKRZ)<sup>3</sup> prior to the SCC and the competition seemed to be a great chance to gain further experience in this field. It is also a great opportunity for us to learn how to work together as a team and how to use our limited time resources to reach our goal.



Figure 1.1: Picture of our Team members and Supervisors

---

<sup>1</sup><http://www.hpcadvisorycouncil.com/events/2016/isc16-student-cluster-competition/img/logos/isc.png>

<sup>2</sup>[http://www.hpcadvisorycouncil.com/events/2016/isc16-student-cluster-competition/img/common/HPC\\_Advisory\\_logo.png](http://www.hpcadvisorycouncil.com/events/2016/isc16-student-cluster-competition/img/common/HPC_Advisory_logo.png)

<sup>3</sup><https://www.dkrz.de/>



## 1.1 Task and Rules

*Author: Jesko Regenthal*

*Philip Gawehn*

Twelve teams participated in the Student Cluster Competition at the ISC 16. A SCC team consists of six undergraduate team members and up to two advisers. The teams that took part in the competition came from all over the world: five teams from Asia, three teams from the US, one team from South Africa and three teams from Europe. Each team configures and optimizes its own small cluster provided by a sponsor to achieve the best results while staying under the power limit of 3kW. In previous years exceeding the 3kW resulted in a decrease of the application's score. This year all not yet submitted results were deleted if teams exceeded the power limit. Therefore, the teams had the opportunity to reach the highest performance by trial and error. After the submission of the HPCC benchmark result, it was no longer allowed to change the hardware setup, even rebooting nodes of the cluster was not allowed to ensure fair competition. Therefore, each team had to choose their hardware configuration carefully. The only exception to this rule was made for the surprise task on the last day.

The task was to run two applications, one we were already familiar with (WRF, which is described in Section 4.2) and a new one (CloverLeaf, which is described in Section 4.4), in less than an hour of time and with the lowest power consumption possible. We were allowed to exchange, remove or add hardware components for these tasks.

The teams compete in three separate categories:

- High Performance Linpack (HPL):

The team with the highest HPL score wins.

- Best overall performance:

All benchmarks and applications are taken into account and an overall score will be determined based on the following criteria:

- 10% HPCC
- 80% application runs
- 10% team interviews

- Fan favorite:

The team getting the most votes wins. Competition visitors and supporters from home can vote for the different teams.

Most of the benchmarks and applications were known beforehand and CloverLeaf (Section 4.4) was the only surprise-application this year.

## 1.2 Booth

*Author: Jesko Regenthal*

*Philip Gawehn*

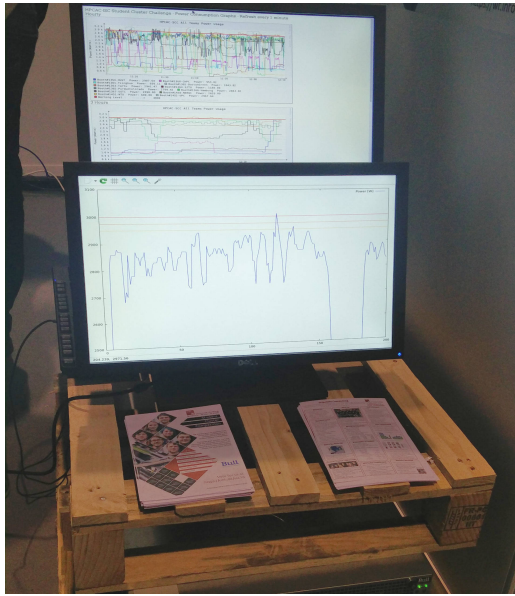


Figure 1.2: Our node monitoring setup

For the competition each team has a booth of 3x3m in which they have to set up their cluster and work in. This means - with multiple nodes, six team members (including laptops), displays and additional hardware for monitoring, cables and switches - space management is of the utmost importance. Our booth is visible in the picture on the left side of Figure 1.3.

To save space during shipping we did not use a rack to mount our two nodes during the competition, instead we stacked them on top of each other using EUR-pallets and small paperboxes as dividers (Figure 1.2).



Figure 1.3: Our booth (left), booths of multiple teams (right)

# 2 System Configuration

*Author: Michael Straßberger  
Julian Frangopoulos*

## 2.1 Hardware Overview

Like last year we obtained our cluster nodes from Bull SAS as our sponsor. We wanted two compute nodes with the following configuration.

- 2 Xeon E5-2680 v4 (14 C / 28 T 2.4 GHz)
- 256 GB DDR4 RAM
- 4 x NVIDIA Tesla K80
- Mellanox InfiniBand, EDR 100 Gbit/s
- 2 x 480 GB Samsung SSD, SATA 6 Gbit/s

This configuration is power efficient, because of the missing Ethernet and InfiniBand switches. Therefore we connected the nodes directly with each other. The K80s consume about 300+ Watts per card. This results in a total of 2400 W (8x300) for the GPUs only. The remaining 600 Watts have been used for CPUs and other components. We underestimated the need for these components. Despite the fact that the PSU has an efficiency of around 94-96%, which we had not consid-

ered in our first calculations. Therefore we had to use power-management features of the K80s and CPUs to fulfill the power limit of 3kW. With our PSU efficiency in mind, we had around 2760 Watts for our components which we need to distribute efficiently, to get the best performance of the system. As a case, we used the Supermicro SYS-1028GQ-TR seen in Figure 2.1. The case features three places for accelerator cards in

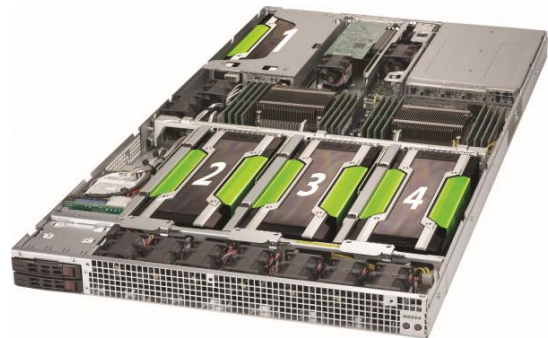


Figure 2.1: SYS-1028GQ-TR from Supermicro <sup>1</sup>

---

<sup>1</sup>[https://www.supermicro.com/a\\_images/products/SuperServer/1U/SYS-1028GQ-TR.jpg](https://www.supermicro.com/a_images/products/SuperServer/1U/SYS-1028GQ-TR.jpg)

the front and 1 in the back. For our air cooling, it contains nine powerful cooling fans. Of which seven were in the front and 2 in the back for the fourth accelerator card.

### **2.1.1 Thermal Problems of the Chassis**

The 1U chassis had some significant disadvantages in comparison to bigger cases. The air cooling performance was not capable of keeping the GPUs at a moderate temperature of about 45 degrees Celsius. The fourth accelerator card in the back of the case was around 80 degree Celsius under heavy load. Since the fourth card was at high temperatures, it reduces its power efficiency to some extent which was relevant for our task of keeping the power consumption under 3kW. After the competition, we calculated with the help of some temperature resistance coefficient formulas<sup>2</sup>, that we had an estimate of 10 to 15% power consumption increase due to the high temperatures in the acceleration cards. Another disadvantage of the high heat dissipation of the acceleration cards in the front was that they heated up the CPU. We got around 70 degrees Celsius at all cores, even when they were under-clocked and mostly idled. Summarized we lost around 5 to 10% of the nodes performance to these thermal problems.

---

<sup>2</sup>[https://en.wikipedia.org/wiki/Temperature\\_coefficient](https://en.wikipedia.org/wiki/Temperature_coefficient)

## 2.2 Operating System



Figure 2.2: CentOS logo <sup>3</sup>

We used CentOS as our Cluster Operating System. First, we discuss what OS we considered to install and under what conditions we preferred CentOS. Then we will explain which software we installed and showcase some configuration snippets and setup

scripts which we used.

### 2.2.1 Choosing the Appropriate OS

We wanted to choose between two operating systems that were suitable for our cluster: CentOS and Qlustar. We preferred CentOS over Qlustar, because of the following advantages:

- More control over our system after configuring most of the applications manually
- Many people are using CentOS for their cluster, therefore the configuration will be supported through community documentations and HowTos.
- In case of errors the overview over the system will help us to fix errors faster

But there were some disadvantages, too. Of course, the configuration took longer with CentOS, so we started the configuration before we had the final hardware. We used a temporary test system on a virtual machine to improve the system configuration gradually.

### 2.2.2 Software Configuration

After choosing CentOS as our operating system, we had to discuss which job scheduler we want to install. Since we already had plenty experience with Slurm, we decided to use it instead of looking for alternatives. To maintain dependencies and different library versions we installed Spack. We explain advantages of Spack in Section 2.4.

---

<sup>3</sup><https://s3.amazonaws.com/awsmpl-logos/centos.png>

## Slurm



Figure 2.3: Slurm logo <sup>4</sup>

Slurm was very helpful at the preparation phase of our cluster. We were able to schedule benchmarks at night to ensure availability at daytime for application testing. Also we could ensure that only one application is running at the same time. Thanks to Slurm we kept the influence on running applications to a minimum. The accounting back-end of Slurm made it easy to compare run-time of applications which did not come with sufficient log output of their performance. Slurm job files ensured to have reproducible environments for our application runs, so we were able to compare them more efficiently.

## Module System



Figure 2.4: Spack logo <sup>5</sup>

In order to find a feasible system for our modules, we first looked at the implementation of the team that attended last year. Their module system consisted of shell scripts which compiled and created module files for various libraries and versions. Their approach had some flaws:

- Manual adjustments to newer versions
- Difficult maintainability

Though their shell scripts offered a higher customization and control over the module system, we decided to use Spack as our abstraction layer over the environment module system. Spack is a simple system, that can be used without caring of dependencies of each module. We thought Spack would be the best solution because of its simplicity and low maintenance effort.

---

<sup>4</sup>[https://upload.wikimedia.org/wikipedia/commons/1/1d/Slurm\\_Workload\\_Manager.png](https://upload.wikimedia.org/wikipedia/commons/1/1d/Slurm_Workload_Manager.png)

<sup>5</sup><https://github.com/LLNL/spack/blob/develop/share/spack/logo/spack-logo-text-64.png>

## 2.3 Filesystem

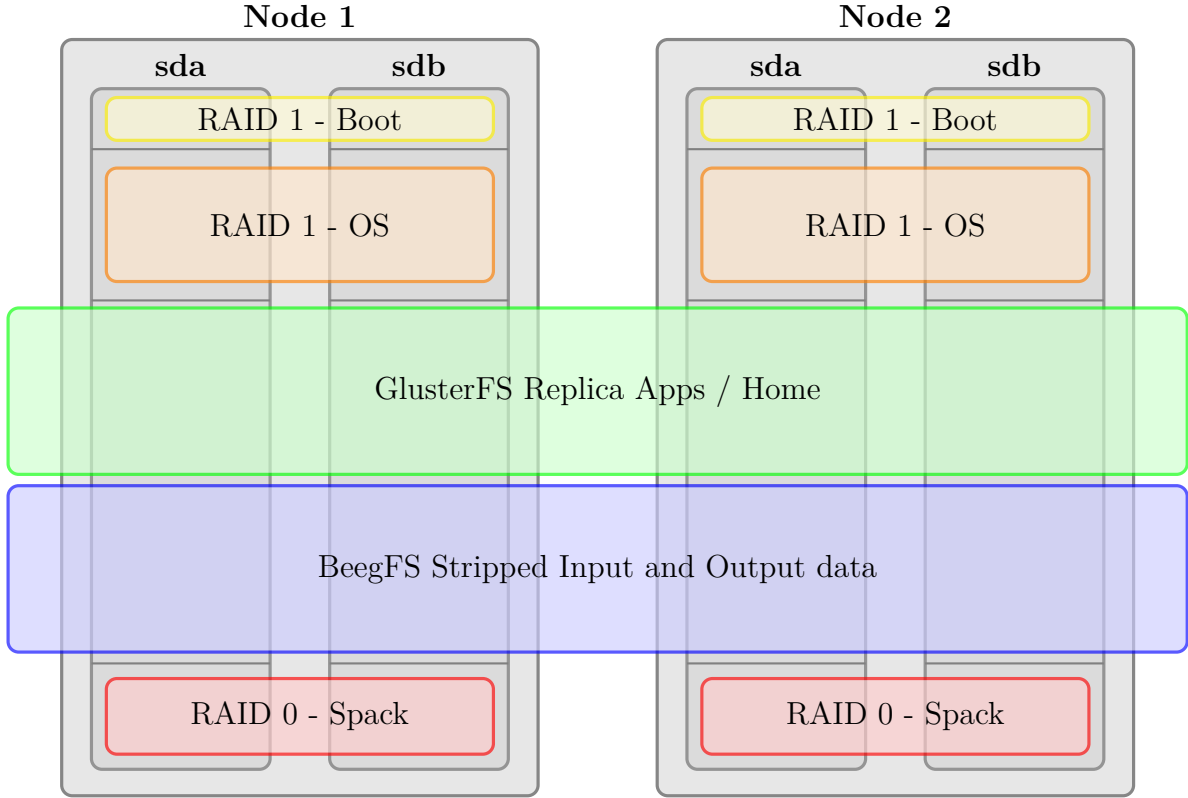


Figure 2.5: Filesystem structure of our two compute nodes

Each node had two Samsung SSDs. We decided to use them with different RAID configurations to get a good tradeoff between performance and reliability. We used a RAID 1 (replication) configuration for our boot and system partition to maintain operation even if an SSD drive fails during the competition. The Partition, in which we stored the modules maintained through Spack, were configured as a RAID 0. We synchronized the two nodes with `rsync` on a daily basis, to maintain consistency. There exist better solutions than this, but for our case, it was far easier to write a Cron job which executed the `rsync` command. We used the remaining storage for our data RAID 0 configuration. On this RAID 0 partition, we created GlusterFS and BeeGFS filesystems. To achieve redundant home directories and application directories, we used GlusterFS, which automatically replicated the data on both nodes. To create a big storage usable for parallel I/O of applications, we used BeeGFS with striped data over all four SSDs. We discuss advantages and disadvantages later.

### 2.3.1 GlusterFs



Figure 2.6: GlusterFS antmascot <sup>6</sup>

GlusterFS is a distributed filesystem that can manage/create replicated and striped storage. It is a kind of software RAID controller over the network. We stored application data in our GlusterFS instance to have replication across our two nodes. The replicated storage was used, because we wanted to have the chance to remove one node from our cluster if it fails or for the surprise challenge at the end of the competition. Writing and reading in GlusterFS needs far more CPU resources than BeeGFS. We missed measuring the differences. On a later test run of WRF the GlusterFS process used about 12 Cores. Therefore GlusterFS was not suitable for our input and output data of the

applications. We kept GlusterFS for our compiled applications because of its enterprise grade reliability.

### 2.3.2 BeeGFS



Figure 2.7: BeeGFS Logo <sup>7</sup>

BeeGFS was developed at the Fraunhofer Institute for Industrial Mathematics. Its primary purpose is for HPC storage solutions. It creates striped storage to increase maximum throughput in HPC applications. Like GlusterFS, it acts like a RAID 0 controller over the network. We use the BeeGFS storage for our application I/O since it was able to reach 2.4 GB/s total throughput. This throughput is possible because BeeGFS takes advantage of the InfiniBand interface of the cluster. To get InfiniBand support

for our interface we had to install specific kernel modules and management packages in CentOS. Since our SSDs interface has been SATA3, its maximum interface speed is 6 Gbit/s. So our four SSDs have had an accumulated performance of 24 Gbit/s which is 3 GB/s. With this setup, we achieved about 80% of the cumulative performance. Since we focussed on application performance and the storage solution satisfied our need in I/O performance we made no further adjustments to BeeGFS.

<sup>6</sup><https://www.gluster.org/images/antmascot.png?1458134976>

<sup>7</sup>[http://www.beegfs.com/content/wp-content/uploads/pics/beegfs-logo/BeeGFS\\_Logo\\_370x269.png](http://www.beegfs.com/content/wp-content/uploads/pics/beegfs-logo/BeeGFS_Logo_370x269.png)



## 2.4 Spack Module System



Figure 2.8: Spack logo <sup>8</sup>

In our research on how to manage different versions of libraries and compilers, we discovered a new project called Spack which was originally written by Todd Gamblin. It is a wrapper around environment modules. Spack allows automated installation of virtually every common library or tool used in HPC. If a library or tool is not present in the package base of Spack, it is easy to write an own package file for that. One of the key features of Spack is the automated dependency resolution. In common module systems, it was easy to load the wrong dependency version of a library. With Spack this is not a case any more. Below we chose a selection of installed modules of our cluster:

compilers	clang-3.8.0
	gcc-4.8.5
	gcc-5.3.0
	gcc-6.1.0
	intel-17.0.0
	pgi-16.4-0
MPI implementations	openmpi-1.10.2
	openmpi-1.8.8
	mvapich2-2.2b
math libraries	openblas-0.2.18
	fftw-3.3.4
CUDA libraries	cuda-6.5.14
	cuda-7.5.18

Beside of that, we used some Intel libraries that were not embedded in Spack, e.g. the Intel Math Kernel Library (MKL).

---

<sup>8</sup><https://github.com/LLNL/spack/blob/develop/share/spack/logo/spack-logo-text-64.png>

## 2.5 Installation

We have written some scripts to automate the installation process of our cluster. The script gave us the opportunity to test our system setup aggressively. We were able to "reset" to a working state of the system easily.

### 2.5.1 Template Script for Program Installation

We used this template script as a base for our reproducible installation script for the main components of our cluster.

```
1 function build-<package-name>
2 {}
3 function install-<package-name>
4 {}
5 function configure-<package-name>
6 {}
7 function postinst-<package-name>
8 {}
9 # Init
10 # Make sure only root can run our script
11 if [ "$(id -u)" != "0" ]; then
12     echo "This script must be run as root" 1>&2
13     exit 1
14 fi
15 build-<package-name> ()
16 install-<package-name> ()
17 configure-<package-name> ()
18 postinst-<package-name> ()
```

Listing 2.1: Template script

### 2.5.2 Slurm Installation

For an easy and fast installation of Munge, we decided to use an existing Munge key. In a real cluster environment, this should be avoided for security reasons, but for our setup that was secure enough. The installation of Slurm afterwards was kind of straight forward.

```

1 #!/bin/bash
2 # Configuration
3 MUNGE_RELEASE_URL="https://github.com/dun/munge/releases/download/munge
   -0.5.12/munge-0.5.12.tar.xz"
4 SLURM_RELEASE_URL="http://www.schedmd.com/download/latest/slurm-15.08.10.
   tar.bz2"
5 #####
6 # DANGER ZONE. Don't Modify anything after this
7 # unless you know what u're doing
8 #####
9 function build-munge
10 {
11     wget ${MUNGE_RELEASE_URL}
12     echo "Building Munge"
13     rpmbuild -tb --clean munge-0.5.12.tar.xz
14     echo "Installing munge"
15     rpm --install ~/rpmbuild/RPMS/x86_64/munge-*
16 }
17 function configure-munge
18 {
19     echo "Running Configuration of munge"
20     chmod 0700 /etc/munge/ -R
21     chmod 0711 /var/lib/munge/ -R
22     chmod 0755 /var/run/munge/ -R
23     cp munge.key /etc/munge/munge.key
24     echo "Start munge daemon"
25     systemctl start munge
26     systemctl enable munge
27 }
28 function build-slurm
29 {
30     wget ${SLURM_RELEASE_URL}
31     rpmbuild -tb --clean slurm-15.08.10.tar.bz2
32     rpm -i ~/rpmbuild/RPMS/x86_64/slurm-15.08.10-1.el7.centos.x86_64.rpm \
33         ~/rpmbuild/RPMS/x86_64/slurm-devel-15.08.10-1.el7.centos.x86_64.
34         rpm \
35         ~/rpmbuild/RPMS/x86_64/slurm-munge-15.08.10-1.el7.centos.x86_64.
36         rpm \
37         ~/rpmbuild/RPMS/x86_64/slurm-plugins-15.08.10-1.el7.centos.x86_64
38         .rpm \
39         ~/rpmbuild/RPMS/x86_64/slurm-sjobexit-15.08.10-1.el7.centos.
40         x86_64.rpm \
41         ~/rpmbuild/RPMS/x86_64/slurm-sjstat-15.08.10-1.el7.centos.x86_64.
42         rpm \
43         ~/rpmbuild/RPMS/x86_64/slurm-torque-15.08.10-1.el7.centos.x86_64.

```

```

39     rpm \
        ~/rpmbuild/RPMS/x86_64/slurm-perlapi-15.08.10-1.el7.centos.x86_64
        .rpm
40 }
41 function configure-slurm
42 {
43     echo "Configure Slurm"
44     useradd -u 512 slurm
45     cp slurm.conf /etc/slurm/slurm.conf
46     echo "Start Slurm daemon"
47     systemctl start slurm
48     systemctl enable slurm
49 }
50 # Init
51 # Make sure only root can run our script
52 if [ "$(id -u)" != "0" ]; then
53     echo "This script must be run as root" 1>&2
54     exit 1
55 fi
56 echo "Installing dependencies for munge and slurm"
57 yum -y -q install wget rpm-build bzip2-devel openssl-devel zlib-devel \
58     gcc readline-devel pam-devel perl-ExtUtils-MakeMaker \
59     perl-Switch
60 build-munge
61 configure-munge
62 build-slurm
63 configure-slurm

```

Listing 2.2: Reproducible Slurm installation script

### 2.5.3 BeeGFS Installation

This script unifies the installation process for management, storage, and meta data nodes. Therefore we did not need to distinguish between the node types on re-installation of the cluster. It also took care of generating service and storage IDs specific for each node based on the hostname naming convention. This step was necessary to automate the installation and configuration of BeeGFS

```

1 #!/bin/bash
2 # Configuration
3 MANAGEMENT_NODE="sccnode1"
4 BEEGFS_MAIN_DIR="/data"
5 BEEGFS_MGMTD_DIR=${BEEGFS_MAIN_DIR}/beegfs_mgmd
6 BEEGFS_META_DIR=${BEEGFS_MAIN_DIR}/beegfs_meta
7 BEEGFS_STORAGE_DIR=${BEEGFS_MAIN_DIR}/beegfs_storage
8 #####
9 # DANGER ZONE. Don't Modify anything after this
10 # unless you know what u're doing
11 #####
12 SERVICE_ID="${HOSTNAME//[0-9]/}"
13 STORAGE_ID=10"${HOSTNAME//[0-9]/}"
14 function install
15 {
16     yum -y -q install wget gcc
17     wget -O /etc/yum.repos.d/beegfs-rhel6.repo http://www.beegfs.com/release
18         /latest-stable/dists/beegfs-rhel6.repo
19     yum -y -q install beegfs-mgmd beegfs-meta beegfs-storage beegfs-client
20         \
21             beegfs-helperd beegfs-utils kernel-devel
22 }
23 function configure
24 {
25     echo -n "Check if we're the MANAGEMENT_NODE..."
26     if [[ $HOSTNAME = $MANAGEMENT_NODE ]]; then
27         echo "YES"
28         /opt/beegfs/sbin/beegfs-setup-mgmd -p ${BEEGFS_MGMTD_DIR}
29     else
30         echo "NO"
31     fi
32     /opt/beegfs/sbin/beegfs-setup-meta -p ${BEEGFS_META_DIR} -s ${SERVICE_ID}
33         -m ${MANAGEMENT_NODE}
34     /opt/beegfs/sbin/beegfs-setup-storage -p ${BEEGFS_STORAGE_DIR} -s ${
35         SERVICE_ID} -i ${STORAGE_ID} -m ${MANAGEMENT_NODE}
36     /opt/beegfs/sbin/beegfs-setup-client -m ${MANAGEMENT_NODE}
37 }
38 function postinst
39 {
40     echo -n "Check if we're the MANAGEMENT_NODE..."
41     if [[ $HOSTNAME = $MANAGEMENT_NODE ]]; then
42         echo "YES"
43         systemctl start beegfs-mgmd
44         systemctl enable beegfs-mgmd

```

```

42  else
43      echo "NO"
44  fi
45  systemctl enable beegfs-meta
46  systemctl enable beegfs-storage
47  systemctl enable beegfs-helperd
48  systemctl enable beegfs-client
49
50  systemctl start beegfs-meta
51  systemctl start beegfs-storage
52  systemctl start beegfs-helperd
53  systemctl start beegfs-client
54  }
55  # Make sure only root can run our script
56  if [ "$(id -u)" != "0" ]; then
57      echo "This script must be run as root" 1>&2
58      exit 1
59  fi
60  echo -n "Test if Main Directory: "${BEEGFS_MAIN_DIR}" exists..."
61  if [ ! -e $BEEGFS_MAIN_DIR ]; then
62      echo "FAILED"
63      exit 127
64  else
65      echo "SUCCESS"
66  fi
67  install
68  configure
69  postinst

```

Listing 2.3: Reproducible BeegFS installation script

# 3 Benchmarks

## 3.1 HPL

*Author: Julian Frangopoulos  
Rasmus Warrelmann  
Michael Strassberger*

The High Performance Linpack (HPL) is a benchmarking tool for measuring the double precision floating point performance of a cluster. The benchmark consists of solving linear equation systems. HPL is used to determine the peak and the average performance of systems listed in the Top-500 list. At the competition, HPL was a major challenge. There was a trophy dedicated to the highest Linpack score.

### 3.1.1 Choosing the Appropriate Version of HPL

As mentioned in Section 2.1 we used a cluster with GPUs from NVIDIA for accelerating the floating point operations per second (FLOPS). We began to research the web for available public solutions which we then can optimize to our demands. For the old Fermi architecture of NVIDIA accelerator cards, there exists free available source code. After having first runs with this code, we were not pleased with the results. We decided to kindly contact NVIDIA if there exist solutions with already optimized code for our particular architecture. We received a compiled binary from NVIDIA for our architecture. The source code was not open sourced, so we did not get the insights of how they achieved the better performance.

### 3.1.2 Optimization

#### FERMI

We ran the FERMI code with various input data to determine which performs best. A subset of the data we collected is presented in Figure 3.1. Since we did not know at this point that we would be allowed to use the binary from NVIDIA we tried various combinations of libraries and compilers to achieve the best performance. We missed to collect data for the different combinations, but we got the best results with GCC 6.x as compiler and the Intel MKL as the math library. The best performance parameters

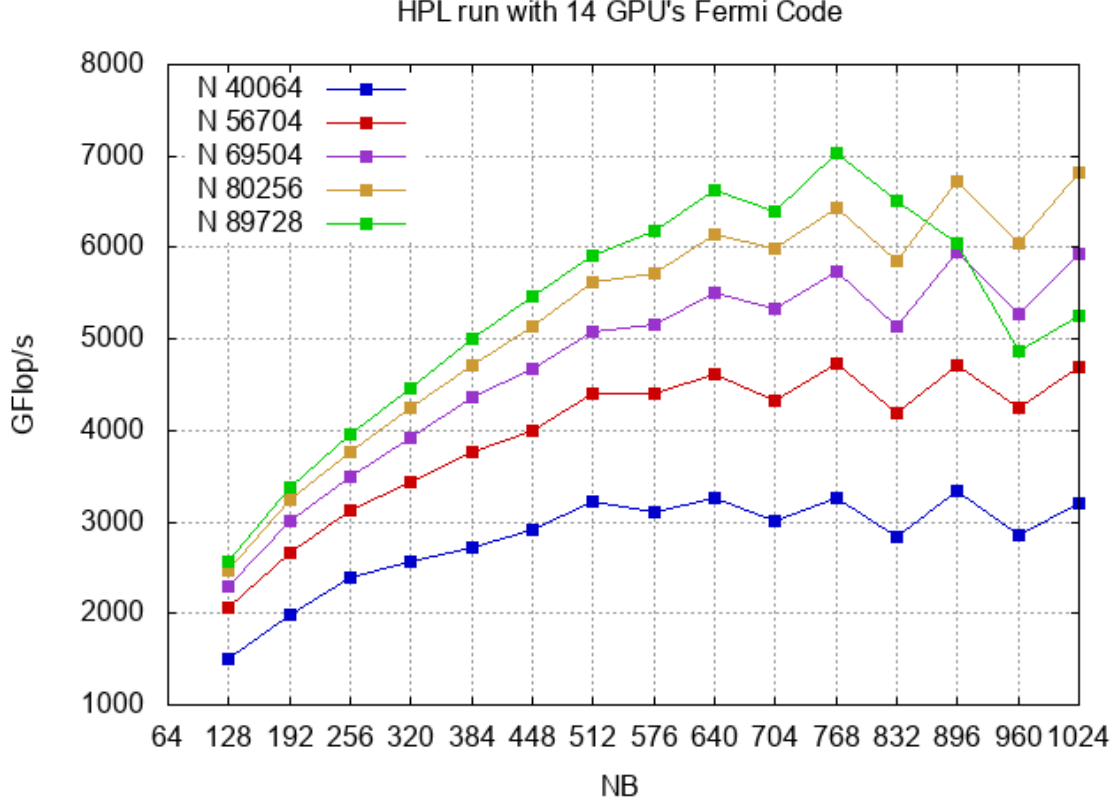


Figure 3.1: Benchmarking different quantities of the FERMI HPL

have been N 89728 and NB 768. After we got permission from the competition staff to use the binary we focused on optimizing the binary instead.

### NVIDIA Binary

To optimize the input data for the NVIDIA HPL binary, we selected different problem sizes with various NBs. After getting decent results of up to 14 TFLOPS, we looked into power consumption. To keep the energy consumption under the given 3 kW power limit, we used various techniques. NVIDIA provides with "nvidia-smi" a tool to set power limits for the accelerator cards. But the limit configured through this tool is only a guide value for the cards. We observed several spikes in the power consumption of the cards. The spikes were about 5 up to 10% above the set power limit. To counter this artefact we tried to throttle the clock speed of our CPUs. We observed that throttling the CPU clock speed also stabilized energy consumption of the acceleration cards. To set this configuration for the cards and CPUs, we used the job script snippet listed in Listing 3.1.



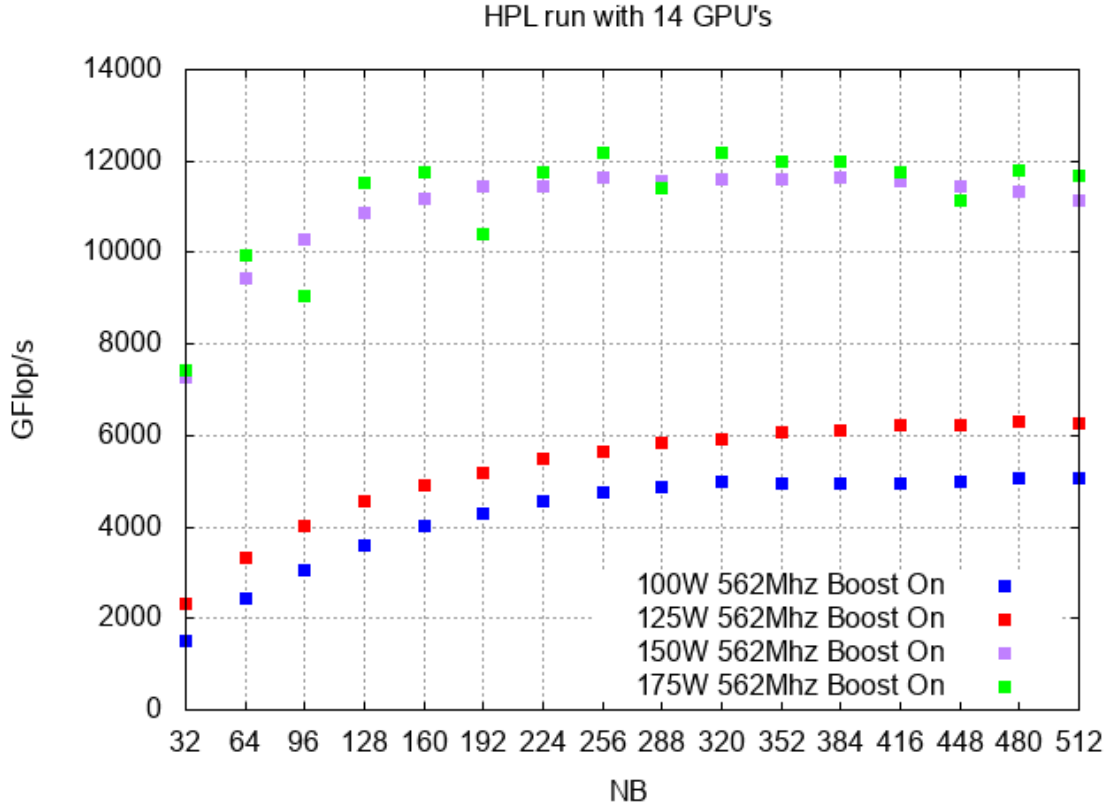


Figure 3.2: Benchmarking different quantities of the NVIDIA HPL binary

```

1  srun -N2 -n2 cpupower frequency-set -d 2300000 -u 2300000
2  [...]
3  srun -N2 -n2 nvidia-smi -pl 158
4  srun -N2 -n2 nvidia-smi -ac 2505,562
5  srun -N2 -n2 nvidia-smi --auto-boost-default=1

```

Listing 3.1: Setting Clockspeed of CPU and GPU

The combination of CPU and GPU throttling resulted in a very stable power consumption. Activating hyper-threading with the corresponding environment variables for OpenMP and Intel MKL increased the performance by 5% by keeping the same power consumption of about 2.95 kW. Results of our testing of different problem sizes, NBs and power limits for the cards are presented in Figure 3.2

### 3.1.3 Results

At the competition, we used our insights of results in Figure 3.2 to fine tune the resulting performance of our cluster. We managed after 27 tries to achieve 12.29 TFLOPS in HPL and staying under the energy limit. The peak power consumption was 2.975 kW, and the average was 2.9 kW. With this result, we broke the world record of the ASC'16 by 0.26 TFLOPS. The cleaned up HPL log is given in Listing 3.2.

```

1 =====
2 T/V              N      NB      P      Q              Time              GFLOPS
3 -----
4 WR01C2R4        146496    192      2      7              170.59              1.229e+04
5 -----
6 ||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=          0.0015874 ..... PASSED
7 =====
8
9 Finished        1 tests with the following results:
10                 1 tests completed and passed residual checks,
11                 0 tests completed and failed residual checks,
12                 0 tests skipped because of illegal input values.
13 -----
14
15 End of Tests.
16 =====

```

Listing 3.2: Our best result which broke previous WR

## 3.2 HPCC

*Author: Julian Frangopoulos*

*Rasmus Warrelmann*

*Michael Strassberger*

The High Performance Computing Challenge (HPCC) benchmark is a collection of seven different benchmarks. They test various aspects of an HPC cluster such as memory, network, FLOPS and MPI implementation. To check these metrics, HPCC consists of the following benchmarks

- High Performance Linpack
- DGEMM
- STREAM
- PTRANS - Parallel Transpose of Matrices
- Random Access
- Fast Fourier Transformation
- Communication Bandwidth and Latency

Since our cluster consisted of accelerator cards from NVIDIA, we needed to evaluate what components of HPCC can be accelerated with our K80s. The most obvious benchmark is the HPL part of HPCC. We discuss our attempts of integrating GPU ready HPL components into HPCC in Section 3.2.1.

### 3.2.1 Optimizing HPCC

Most of the elements of HPCC are CPU intensive tasks. The main options of HPCC to tune performance are the problem size  $N$  and  $NB$ . Choosing the right FFT library and different MPI implementations have a significant impact on the HPCC score. Intel provides with Intel MKL a mathematical library with FFT support. MKL takes advantage of all advanced CPU features to increase the overall performance of the FFT operations.

To get an insight what  $NB$  is best fitting for our setup we looked at the specification of our CPUs. For best performance, it is important that the matrices fit in the level 1, 2 and 3 caches of our CPUs. For our model, they have been 32 KiB L1, 256 KiB L2 and

35 MiB L3. We assume that no other process utilizes the level 1 data cache of our CPU core. We now calculate how much floating point values can be stored in the cache and get the square root of it 3.3. The same we also do for the level 2 cache in 3.6 and the level 3 cache in 3.10. The level 3 cache is divided by 14 because the 14 cores of the CPU share those 35 MiB level 3 cache.

$$32KiB * 1024 = 32768B \quad (3.1)$$

$$32768B/8B = 4096 \quad (3.2)$$

$$\sqrt{4096} = 64 \quad (3.3)$$

$$256KiB * 1024 = 262144B \quad (3.4)$$

$$262144B/8B = 32768 \quad (3.5)$$

$$\sqrt{32768} \approx 181.02 \quad (3.6)$$

$$32MiB/14 = 2.5MiB \quad (3.7)$$

$$2.5MiB * 1024 * 1024 = 2621440B \quad (3.8)$$

$$2621440B/8B = 327680 \quad (3.9)$$

$$\sqrt{327680} \approx 572.42 \quad (3.10)$$

With the previous calculations made we now have a lower and upper bound for our NB. The best value for NB must be between 64 and 512 (572 rounded to the next lower multiple of 64). To get a higher resolution, we tested the NB with steps of 32 in Figure 3.3

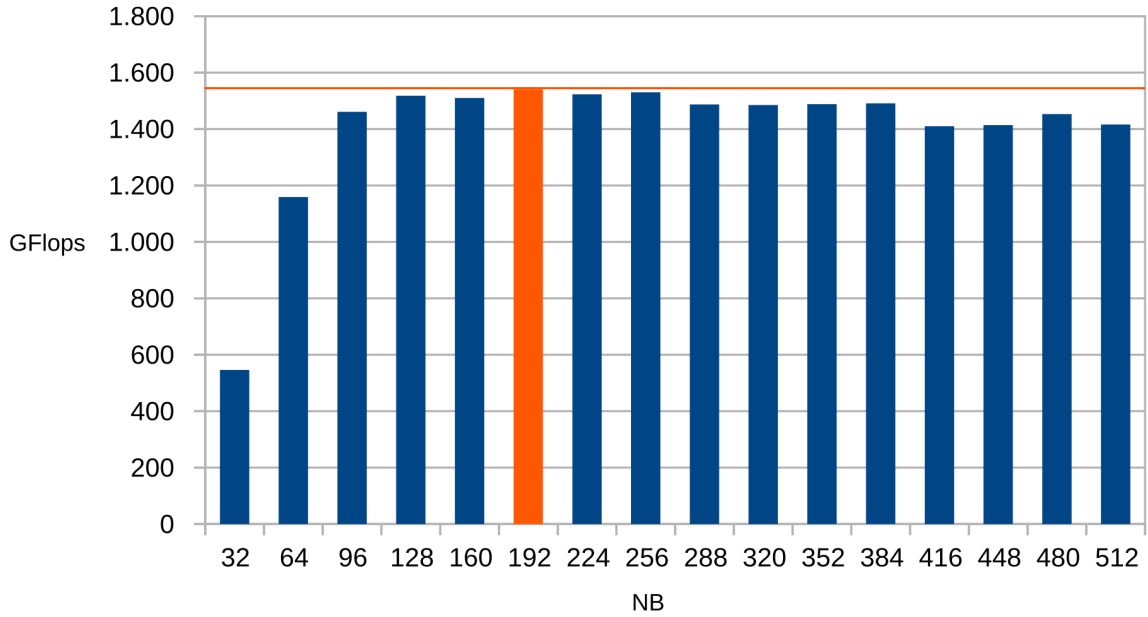


Figure 3.3: Comparison of NB and resulting GFLOPS

### Trying to merge HPCC with the GPU-accelerated HPL

One of the most time consuming challenges was to elaborate the possibilities to use our GPU in the HPL part of HPCC. Since there was no accessible existing version of HPCC that was ready for GPU computing, we decided to look into that task ourselves. We thought about many ways to archive this:

1. Substitute the CPU source code with available GPU code
2. Inserting provided compiled binary of NVIDIA into HPCC
3. Rewrite HPL part to use GPU accelerated libraries

### Substitution of CPU Source code

We found several GPU supporting implementations of HPL on [github.com](https://github.com). But we had some different problems with each of them: At the time of trying to compile the source code, we missed some essential libraries used by the implementation. On the other side, the HPL implementation of HPCC used the header file and some variables of HPCC. So using another HPL version caused several compatibility issues and we would have to rewrite parts of the application. Also some of the other benchmarks of

HPCC used the data structures of the HPL implementation. We tried to use the HPL code optimized for the FERMI architecture, but this would result in rewriting or altering many parts in the HPCC application. Because of the arising deadline of the competition we decided to discard this idea.

### **Inserting pre-compiled NVIDIA binary**

Because we already had a decent running pre-compiled binary from NVIDIA, which was very good optimized for our GPU architecture, we thought of inserting the binary into HPCC. That would be realized through calling the main function of the NVIDIA binary. To do so we first had to investigate how deeply code of HPL is reused in the other benchmarks of HPCC. After a short time we realized that this would be very difficult to archive. Later on at a meeting we figured out that we did not have any time for this idea either.

### **Rewriting HPL with GPU libraries**

Another possibility of achieving GPU accelerated HPCC was to use existing libraries to substitute existing function calls in the source code of HPCC. After some short tests with some wrappers for these HPCC function calls, we have managed to get compiled binaries which used GPUs but did not compute on them properly. After another short test, we dropped this idea and started to begin optimizing for CPU instead.

### **Using the optimized STREAM-Benchmark**

The team from last year created an accelerated version of the STREAM Benchmark using assembler. We tried out this version on our cluster and experienced a way better performance than with the standard implementation.

# 4 Applications

## 4.1 WRF

*Author: Philip Gawehn  
Jesko Regenthal*

The *Weather Research and Forecasting Model* (WRF)<sup>1</sup> provides one of the newest numerical models for climate research. WRF is widely used for forecasting and in science. The development began in the second half of the 1990's. During its continuous development for about 20 years, it has evolved to an advanced software. The "National Weather Service", the "US Military" and other meteorological organisations use WRF productively.

WRF contains two different dynamic solvers. First the *Advanced Research WRF* (ARW) core and second the *Nonhydrostatic Mesoscale Model* (NMM) core. For the competition, we had to use the ARW core which provides options to run calculations based on given real world data. To prepare data used by WRF (real case), the *WRF Preprocessing System* (WPS) is also needed.

Because we did not use the NMM core, we reference ARW as WRF in this Section.

### 4.1.1 WRF Version and Dependencies

There are two releases of WRF each year, a new major release each February and a maintenance update around August. To achieve the best performance we used the newest available release. Our first attempts to compile and run WRF were made with version 3.7.1, while later runs were made with version 3.8. About two weeks before the competition there was an announcement that we had to use the older version 3.6.1. This was no major problem as our prepared script for building and testing (described in Section 4.1.2) needed only minor adjustments.

WRF is written in Fortran and C and depends on the following libraries (v3.6.1):

- NetCDF<sup>2</sup> (Network Common Data Form) v4.1.3:  
A file format for the exchange of scientific data

---

<sup>1</sup><http://www.wrf-model.org/index.php>

<sup>2</sup><http://www.unidata.ucar.edu/software/netcdf/>

- `zlib`<sup>3</sup> v1.2.7:  
A free library for compression and decompression
- `JasPer`<sup>4</sup> v1.900.1:  
A free JPEG 2000 codec reference implementation
- `libpng`<sup>5</sup> v1.2.50:  
The official PNG reference library
- `MPICH`<sup>6</sup> v3.0.4:  
A freely available and portable implementation of MPI

Since more than one year has passed since the release of WRF v3.6.1, we checked the dependencies for updates and moreover, picked the following dependency versions:

- `NetCDF` v4.4.0
- `NetCDF-Fortran` v4.4.4
- `zlib` v1.2.8
- `JasPer` v1.900.1
- `libpng` v1.6.21
- MPI implementation (`MVAPICH` v2.2b or `OpenMPI` v1.10.2)

We added `NetCDF-Fortran` to the dependencies because the Fortran portion was moved to a separate package in v4.2. `MPICH` is interchangeable with any MPI implementation, such as `MVAPICH` or `OpenMPI`. We did not use `MPICH`, as it does not support native `InfiniBand`.

### 4.1.2 How to Build

The compilation of WRF is a time-consuming task, therefore we decided to write a `compile-bash-script`<sup>7</sup>, which is based on an online tutorial<sup>8</sup>.

---

<sup>3</sup><http://zlib.net/>

<sup>4</sup><http://www.ece.uvic.ca/~frodo/jasper/>

<sup>5</sup><http://libpng.org/pub/png/libpng.html>

<sup>6</sup><https://www.mpich.org/>

<sup>7</sup>This script and all other mentioned scripts are included in the `scripts/WRF` directory

<sup>8</sup>[http://www2.mmm.ucar.edu/wrf/OnLineTutorial/compilation\\_tutorial.php](http://www2.mmm.ucar.edu/wrf/OnLineTutorial/compilation_tutorial.php)



To test different compilers and MPI implementations, we wrote a wrapper script which performs the following steps:

- loading the required modules
- set environment variables
- compiling each given configuration

This script includes a settings file containing different installation configurations. To create a configuration, we appended an array in the settings file, as shown below.

```
1 declare -A arr
2 # ...
3 arr[6-enabled]="true"
4 arr[6-compiler]="intel-icc"
5 arr[6-mpi]="mvapich2@2.2b%intel@17.0.0"
6 arr[6-dir]="intel-17.0.0/mvapich-2.2"
7 arr[6-fc]="ifort"
8 arr[6-f77]="ifort"
9 arr[6-cc]="icc"
10 arr[6-cxx]="icpc"
11 arr[6-cp]="20"
12 arr[6-cp2]="19"
```

Listing 4.1: configuration case for a compilation

A configuration is enabled if set to `X-enabled="true"`. `X-compiler` and `X-mpi` should contain a Spack module. If a Spack module did not exist, we passed a specific string which triggered alternate configurations. The compilation needs Fortran compilers (`X-fc` and `X-f77`) and C/C++ compilers (`X-cc` and `X-cxx`). During the compilation, a specific code has to be entered for each configuration. For WRF the code is specified in `arr[6-cp]` and for WPS in `arr[6-cp2]` and in the case of the Intel-ICC compiler describes "Linux x86\_64 i486 i586 i686, Xeon (SNB with AVX mods) ifort compiler with icc (dmpar)" and "Linux x86\_64, Intel compiler (dmpar)", respectively. The compiled WRF binaries will be placed in `X-dir`.

In the settings file, we added a variable for the WRF compile case which was "em\_real" as we had to calculate given real world data.

### 4.1.3 How to Run

WRF takes a configuration file as its input as well as input generated by WPS. To make testing easier, we wrote another script which performs the following preparations:

- downloading real world data (amount changeable through a variable)
- modifying configuration files
- generating WRF input by using WPS

After the download is completed, the configuration files `namelist.input` of WRF and `namelist.wps` of WPS will be updated with the corresponding dates. The downloaded intermediate data is used to generate the input files for WRF.

Executing the run script with the desired number of nodes and processes starts WRF. This run script also uses the settings file and queues runs for all enabled compiler configurations. The generated output files are later used by the `show_output` script.

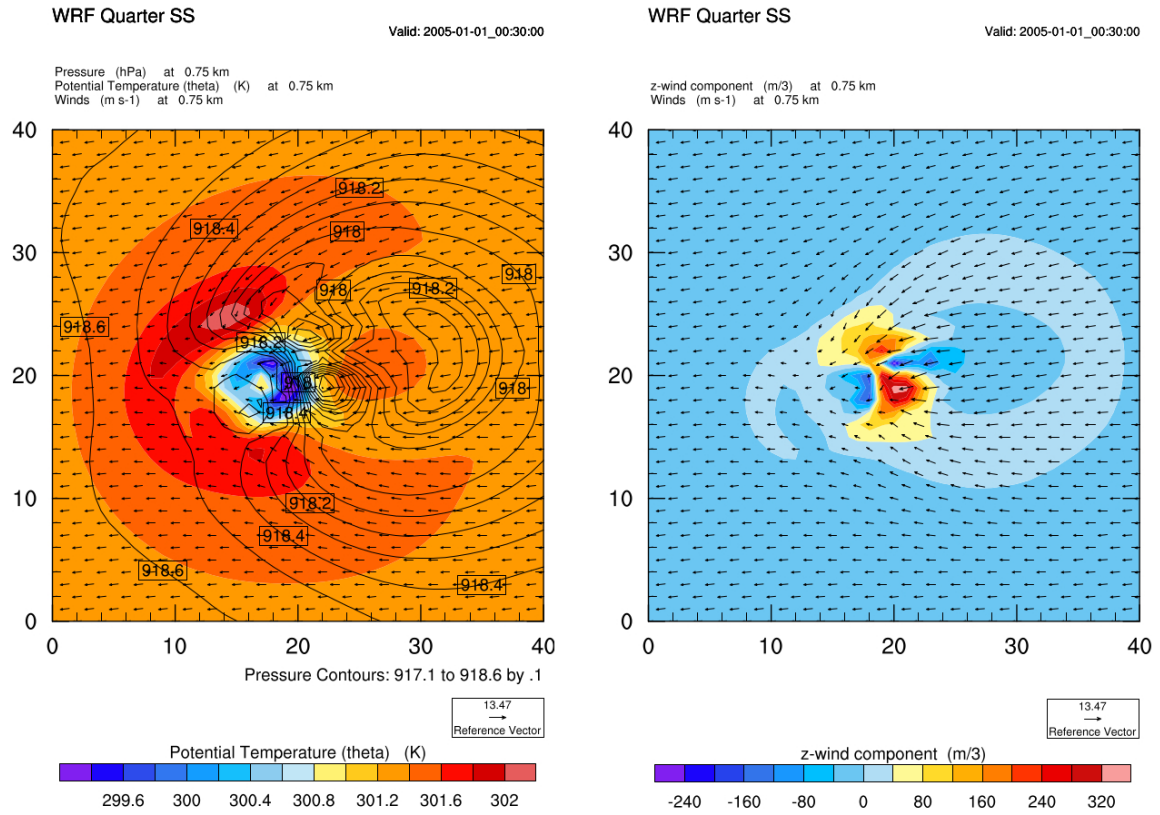


Figure 4.1: An example visualization (test run)

#### 4.1.4 Problems and Further Comments

To use our GPUs, we wanted to use the PGI Accelerator which was the only available GPU accelerated option, but we had problems with it. The PGGroup supplied us with

a license which, for reasons unknown to us, only worked with one specific version of the PGI Accelerator. With a separate free evaluation license, we were able to test other versions. During our tests we found that newer versions of the PGI Accelerator do not work with WRF. We were not able to find a precise reason for this issue. During our research we found other reports of compatibility issues with newer PGI releases. After the announcement two weeks before the competition, that we had to use v3.6.1 of WRF, we focused on further testing this older version for all our compiler configurations. In the end, we did not have the time to experiment further with an older version of PGI to see if it would have worked with our GPUs.

WRF appeared twice during the competition. First as part of the regular "application runs" and second as a "secret application" with the added restriction of using lowest possible power. While the regular application run was completed without errors, we had some problems with the second set of input files.

During our first test run with the second set of input files we noticed some unexpected results. The error log file and some research revealed that the problem was the maximum stack size. To fix this problem, we appended `ulimit -s <NEW_STACK_SIZE>` to our execution script. WRF consumed about one-third of the memory. As we had enough free memory we increased the maximum stack size to 1 GB.

We noticed the second problem after starting our measured execution. WRF printed the following error message:

```
"open_hist_w : error opening wrfout_d01_2016-05-28_00:00:00 for
writing."
```

This error message was confusing from our point of view as WRF created the file, but for some reason could not open it for writing. During the available time we tried to fix this problem with the following measures:

- We checked the file permissions and reset the directory permissions to grant permission for everyone which did not resolve the issue.
- Then we tried to change the file system, so we copied all WRF files and ran a test on a tmpfs, BeeGFS and GlusterFS. Again the problem remained the same, regardless of the file system.
- We checked the checksums of the files multiple times to make sure the files were not corrupted.

- Back in Hamburg we discovered that `export WRFIO_NCD_LARGE_FILE_SUPPORT=1` may have resolved this error. This option has to be set before compiling WRF and would not have helped us in the competition as we did not have the time to recompile WRF.

Due to our issues with the PGI Accelerator we were not able to use our GPUs. Thus we had no problems with the power limitation. During the competition, we stayed below 1,5 kW.

#### 4.1.5 Benchmarks

Before the competition, we benchmarked the different compiler configurations. Figure 4.2 shows the most expressional benchmark comparing the different compiler and MPI library combinations. It shows that the Intel compiler *icc@17.0.0* combined with the MPI library *mvapich2@2.2b* results in the best performance. Figure 4.3 visualizes how different node/process combinations influence the runtime. In the graph, one can notice the better runtime when using 52 cores in total instead of all possible 112 cores in both nodes. In earlier benchmarks, we found that WRF performs better using no more than half of the available cores. We assume the reason for this is that two logical cores share a single floating point unit. Considering this limit of 56 efficiently usable cores per node and subtracting a few cores for other tasks, probably I/O operations, we ended up with a total of 52 cores for efficient usage. We could not identify the reason for the increased performance by leaving a few cores out - we just noticed this fact in our benchmarks (see Figure 4.3).

It would be interesting to see how *icc@17.0.0* in combination with *openmpi@1.10.2* would perform. We were not able to compile OpenMPI using the ICC compiler, so there are no results for this case.

#### 4.1.6 Results

Duration	Nodes/Processes	Status	Compiler	MPI
3957	2/52	success	intel-icc	mvapich2@2.2b%intel@17.0.0

Table 4.1: Our results of day 2

Our first run of WRF on the second day completed in *3957* seconds while running on two nodes, each with 52 processes. As mentioned earlier, we did not finish our

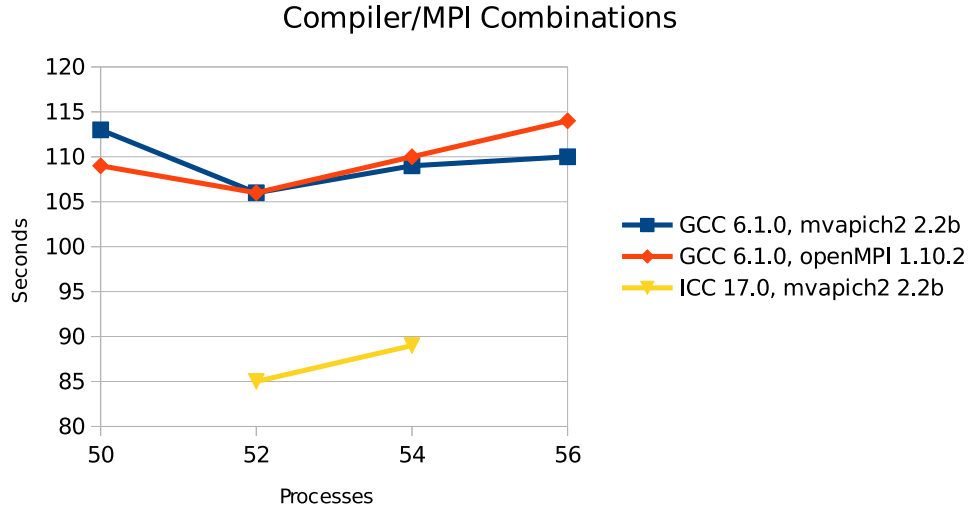


Figure 4.2: Comparison of compiler and MPI library combinations

second run on day 3. Our latest estimate was that the execution would have finished in just over 60 minutes which would have been over the time limit. During the run, we continuously re-evaluated the expected run time and adjusted the CPU frequency to get as close to the 60-minute mark as possible while keeping the power consumption as low as possible. Towards the end, we realised that we underestimated the remaining time in the beginning and did not increase the CPU frequency aggressively enough.

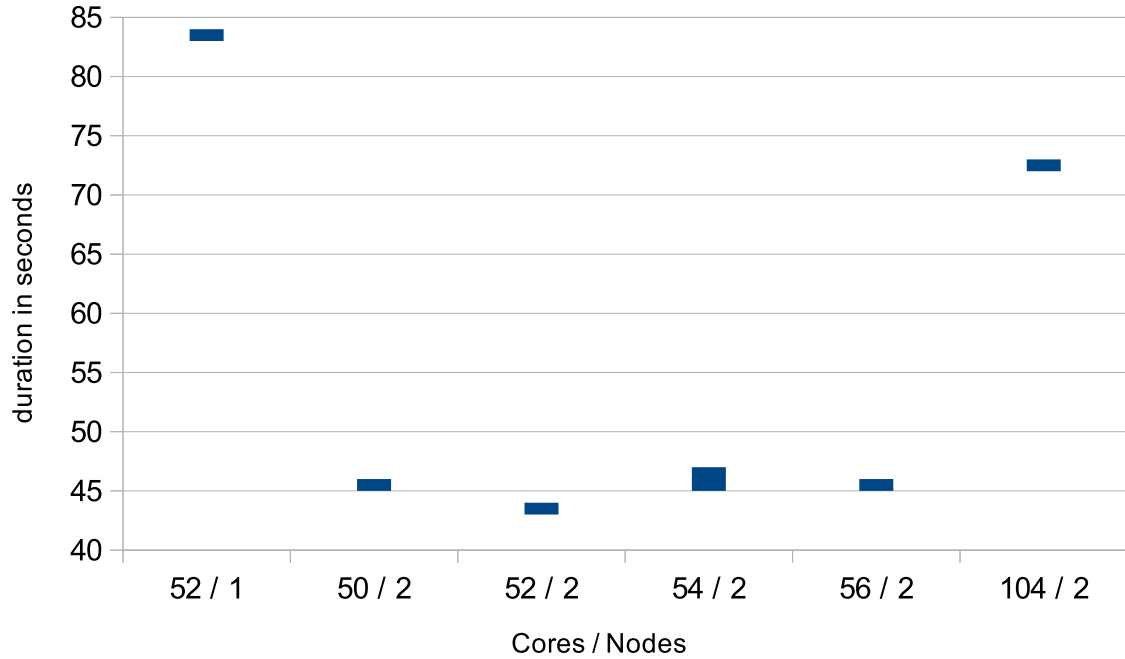


Figure 4.3: Benchmark of *icc@17.0.0* in combination with *mvapich2@2.2b*

## 4.2 Splotch

*Author: Sönke Behrendt*

*Jesko Regenthal*

Splotch<sup>9</sup> is a ray tracing application for visualizing (cosmological) Smoothed-particle hydrodynamics (SPH) simulation data. It can visualize a variety of input data generated by industry standard cosmological simulation applications like Gadget<sup>10</sup>.

### 4.2.1 Dependencies

Splotch is written in C++ and does support OpenMP, MPI, OpenCL and CUDA for parallel execution. Although OpenCL support is still in the code base it is deprecated and should not be used any more. For this reason we focused on CUDA for the GPU support.

The dependencies vary slightly depending whether the calculations should take place on the CPU or GPU.

For the MPI version a MPI library is required. The CUDA version depends on CUDA Toolkit 4 or newer. We did not test it with CUDA 8 as the final version was not released

<sup>9</sup><http://wwwmpa.mpa-garching.mpg.de/~kdolag/Splotch/>

<sup>10</sup><http://wwwmpa.mpa-garching.mpg.de/~volker/gadget/index.html>

in time for the competition.

### 4.2.2 How to Build

Compiling Splotch with the provided makefile is a straightforward process when it is only compiled for CPU support. The only thing we had to enable was MPIIO support which has to be manually compiled before it can be used.

Enabling GPU support required some minor changes. First, CUDA support had to be manually enabled in the makefile and the CUDA library path had to be adapted to work with the package manager Spack. To enable support for the NVIDIA K80 GPUs we had to set the environment variable NVCCARCH with the parameter `-arch=sm_37`. Additionally we used the NVCCFLAGS flag `-use_fast_math` for CUDA.

To be able to test different combinations of compilers and MPI libraries we wrote two install scripts<sup>11</sup>. One which takes the wanted configuration as parameters and one which utilizes the first script and compiles multiple specified versions. This automated the compilation process which saved us time later on. It also made it possible to easily add new configurations.

```
1  $ ./install.sh
2  Usage:
3  Need 4 parameters.
4  ./install.sh MPI CC CUDA OPENCL|CUDA
5  Where MPI, CC and CUDA are Spack module names
```

Listing 4.2: How to use the install script

### SCC'16 Specific Version

For the competition we were unable to use the latest release version 6. Instead we had to use the Git commit *99c9131*<sup>12</sup>. It was required to use this specific commit because parts of the code for the timing summary changed which was used for determining the winner. During the preparation for the competition we periodically checked for new git commits and analysed if our configuration still worked with the newest version. Because of this we already had done our tests with the version specified for the competition and we build it the same way as previous versions which we described earlier.

---

<sup>11</sup>these scripts and all other mentioned scripts in this section can be found in the scripts/Splotch directory

<sup>12</sup><https://github.com/splotchviz/splotch/commit/99c9131>

### 4.2.3 How to Run

Splotch takes a single parameter file as an input. All options are provided in that parameter file. It does not need a specific name or location as you need to pass the name and relative or absolute path to Splotch when you run it. Similar to our install script we wrote a run script which uses the same syntax as the install script to provide some consistency between the usage of the two scripts.

```
1  $ ./run.sh
2  Usage:
3      Need 4 parameters.
4      ./run.sh MPI CC CUDA OPENCL/CUDA
5      Where MPI, CC and CUDA are Spack module names
```

Listing 4.3: How to use the run script

### 4.2.4 Problems

One problem we encountered was high setup times in the CUDA code. In some configurations the CUDA preparation took 90 percent of the time to calculate an image. We have contacted the developers about this problem to check whether this was an expected behavior and for them it was not unusual. Our tests with OpenCL showed a better performance than CUDA. After inspecting the log files and monitoring the GPUs during the runs it became clear that there was no or at the most very little computation done on the GPUs. Therefore the OpenCL performance was roughly equal to our CPU-only runs (see Figure 4.8). After we exchanged some emails with one of the developers it became clear that the OpenCL implementation was deprecated.

Another issue we had was the use of multiple GPUs with Splotch. Even when running a job on multiple GPUs it would just use one and perform a lot slower than it potentially could. This was an error in the Splotch implementation which we managed to solve. It was a rather trivial mistake and shows that the developers did not fully test Splotch on multi-GPU setups. There was no optimization for that case, which may be an explanation for the poor GPU performance that we encountered. After talking to one of the developers about this problem he made similar changes to the Splotch repository like we did to our code <sup>13</sup>. With these changes we were able to speed up the computation time when using multiple GPUs, but still were not able to get a full utilization of the

---

<sup>13</sup><https://github.com/splotchviz/splotch/commit/2240ed5>



GPUs. This might be an issue with the type of problem Splotch solves, which can not be fully parallelized and only parts are calculated directly on the GPU.

We were also only able to compile Splotch with GCC 4.8.5 due to incompatibilities between CUDA 7 and GCC versions later than 4.8.5. Support for the latest GCC compiler 6.x is only available with CUDA 8 which was not released in time for the competition. Using a similar workaround like the one used for Graph500 (described in Section 4.3.3) did not work as the built binaries still not produced any results.

Although much effort has been put into the GPU support for Splotch in the end there were too many problems with the GPU versions and all computations were run on the CPU in the final competition.

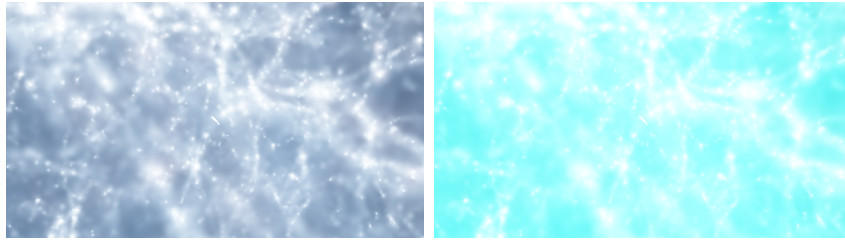


Figure 4.4: Comparison of generated images by CUDA (right) and CPU only (left) runs

Another issue is shown in Figure 4.4. When using CUDA the generated images use a different color palette than when using the CPU only implementation. We did not look into the reason for the color difference between the generated images as we prioritized reducing the GPU setup times and later decided against using GPUs.

### 4.2.5 Benchmarks

In our preparation phase we ran tests with differences in the number of nodes, processes, threads and GPUs. In the following we will show a selection of our performance results.

#### Influence of the number of used GPUs

```
1 % gtx570
2 saving file uni_cuda0008 ...
3 Total wall clock time for 'Splotch': 67.5447s
4 |
5 +- CUDA : 95.53% (64.5253s)
6 | |
7 | +- CUDA : 99.88% (64.4470s)
8 | | |
```

```

9 | | +- CUDA Rendering : 96.90% (62.4522s)
10 | | +- Data copy : 2.25% ( 1.4488s)
11 | | +- Device setup : 0.32% ( 0.2075s)
12 ...
13
14 % K80
15 saving file uni0008 ...
16 Total wall clock time for 'Splotch': 34.5592s
17 |
18 +- Post-processing : 51.14% (17.6728s)
19 +- CUDA : 42.37% (14.6435s)
20 | |
21 | +- CUDA : 66.18% ( 9.6906s)
22 | | |
23 | | +- Device setup : 95.44% ( 9.2487s)
24 | | +- Data copy : 1.68% ( 0.1629s)
25 | | +- CUDA Rendering : 0.47% ( 0.0458s)
26 ...

```

Listing 4.4: Comparison between multiple K80s and a single desktop grade GTX570

During our first tests we wondered about the high amount of setup times for our K80s. To see if this was a problem with the CUDA implementation in general or with the K80s we made a small comparison between multiple K80s on our cluster and a single GTX570 on a desktop machine (Listing 4.4).

After this we looked further into whether the number of utilized GPU had any impact on the setup times. Therefore we ran small test runs with 1, 2, 4, 8 and 16 GPU cores. Figure 4.5 shows that as soon as we use more than 1 GPU (each GPU has 2 cores) the CUDA setup time makes up roughly 50% of the total wallclock time. This demonstrates our main problem with running Splotch on GPUs well. It was a rather small test run which generated only 49 images, but the core message remains the same for larger runs. Spending half of the runtime with setting up the GPUs was not feasible for us, as this decimated all computing benefits of the GPUs over the CPUs.

### Influence of the compiler on performance

To see if the high setup times were related to the used compiler we tested the compiler influence on the GPU performance. As the CUDA implementation was only running with OpenMPI we could not perform tests to see if MVAPICH would have given us better results.

Figure 4.6 shows that using GCC as the compiler results in slightly faster run times.

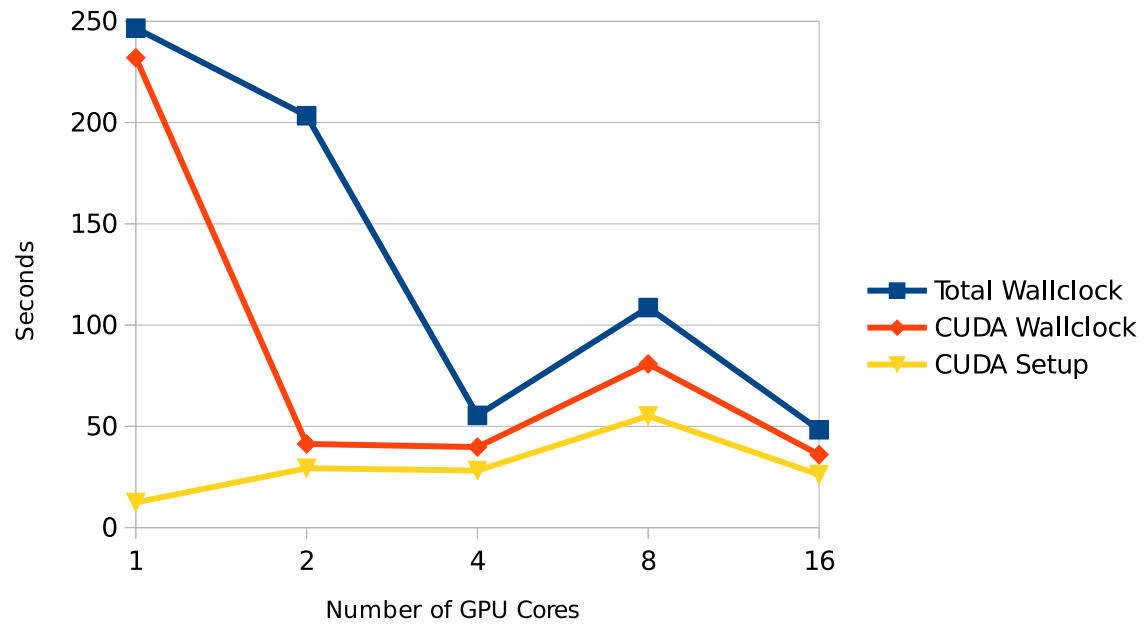


Figure 4.5: Influence of the number of used GPU Cores on the CUDA setup time

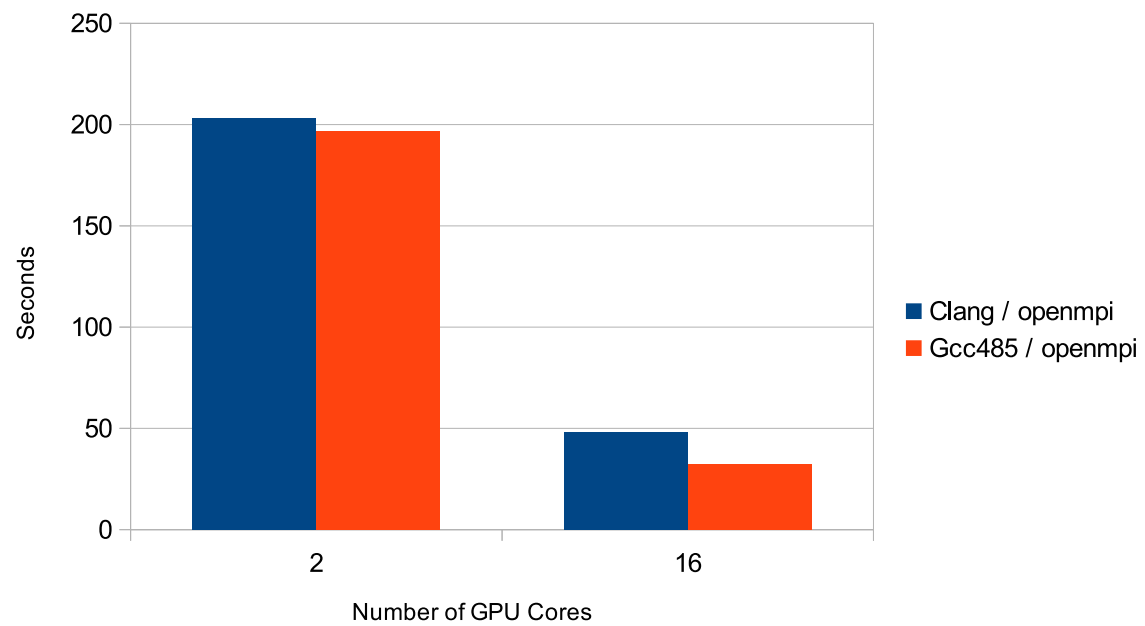


Figure 4.6: Comparison between Clang 3.8 and GCC 4.8.5

Further analysis into the influence of different configurations and input files was not done as the high setup times disqualified the CUDA implementation.

## CUDA vs OpenCL Performance

After testing the influence of the number of utilized GPU cores we wanted to compare the performance of the CUDA and OpenCL implementations.

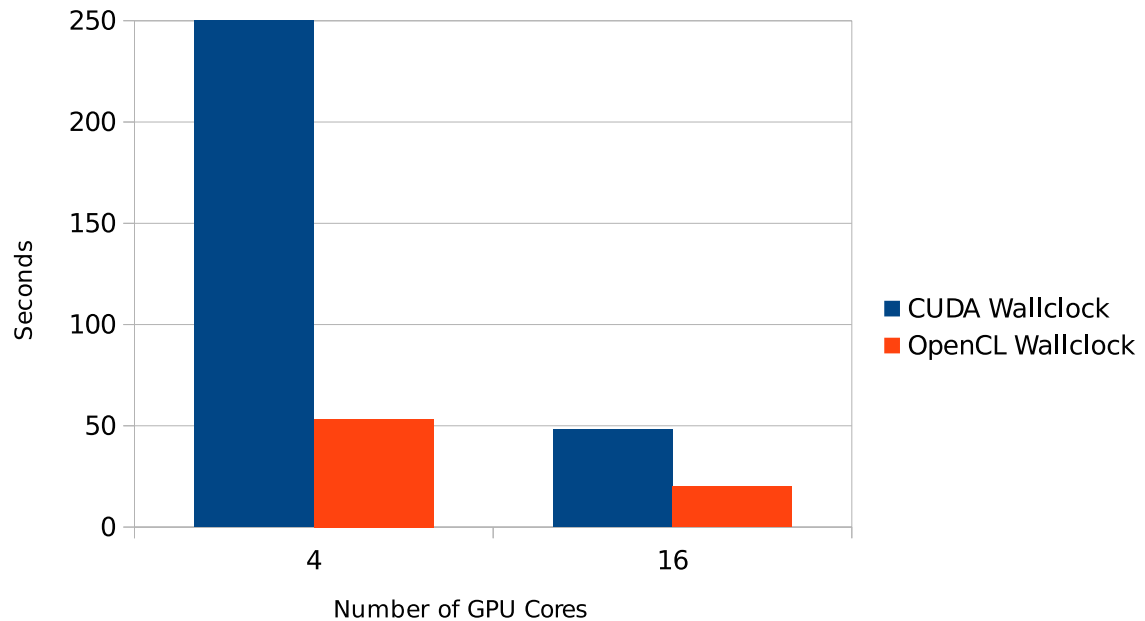


Figure 4.7: Comparison between OpenCL and CUDA run mode

Figure 4.7 suggests that OpenCL is performing a lot better than CUDA. After examining the output file for a reason for this unexpected behavior we found that the OpenCL version was not using the GPUs at all, effectively making it the same as a CPU only run.

```
1  -----
2  Summary of timings
3  -----
4
5  Times of GPU:
6  Copy (secs) : 0
7  Transforming Data (secs) : 0
8  Load OpenCL kernel (secs) : 0
9  Filter Sub-Data (secs) : 0
10 Rendering Sub-Data (secs) : 0
11 Combine Sub-image (secs) : 0
12 OpenCL thread (secs) : 0
13 -----
```

Listing 4.5: Excerpt from a OpenCL log file

This becomes also clear if we compare the runtime of a OpenCL run with a CPU run with a similar or identical node/process/thread combination. (Figure 4.8)

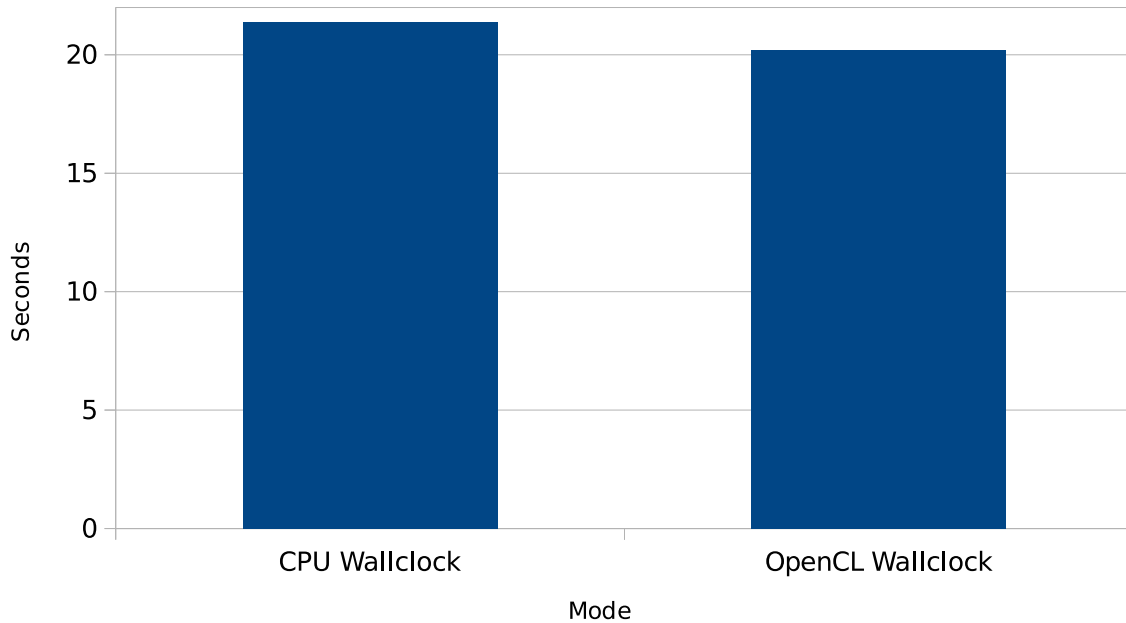


Figure 4.8: Comparison between OpenCL and CPU run mode

### CPU vs GPU Performance

At some point we came to the conclusion that the CUDA and the OpenCL implementations should both not be used (see Figure 4.5 and Figure 4.8), at least not without doing extensive debugging and code tuning first. The official statement regarding changes to the source code was that we would need to send in our changes for approval. At this stage in the preparation we decided to focus on the CPU implementation, as there was not enough time to re-implement the CUDA functions. The risk of not being allowed to use our modifications was another factor.

Initially we focused on the GPU support for Splotch to achieve the best performance and did not spend much time on analyzing every possible compiler and MPI library combination due to their small impact on overall performance and incompatibilities with CUDA. When we decided to not use the GPU we continued to use OpenMPI and GCC as the compiler. We noticed only later when analyzing the data that a combination of OpenMPI and Clang was slightly faster in some cases as shown in 4.10.

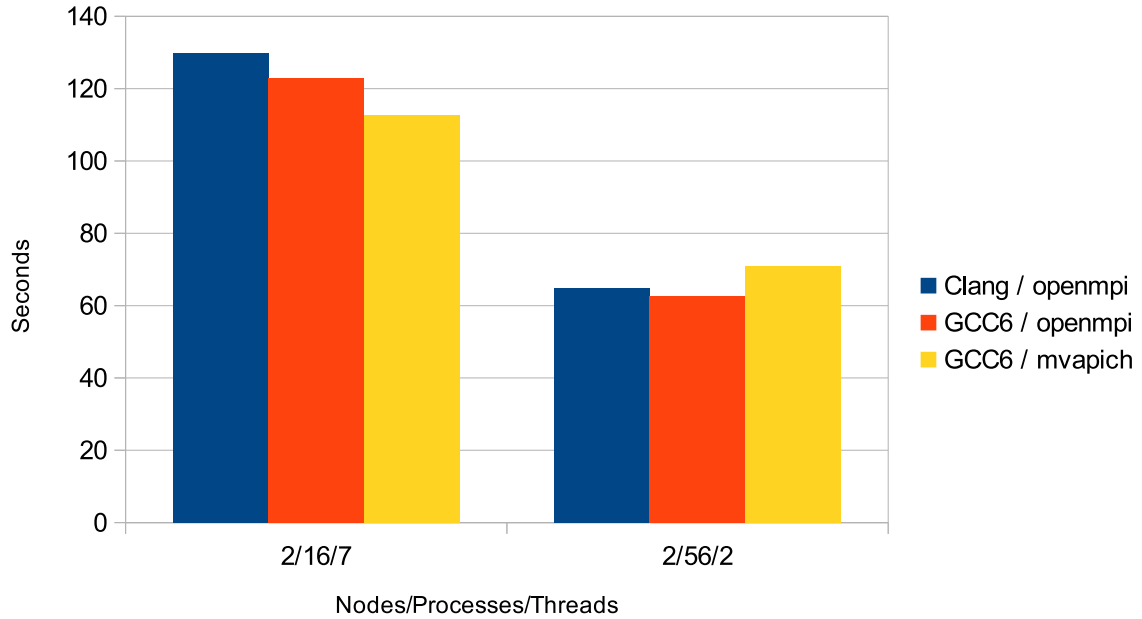


Figure 4.9: Comparison compiler, MPI library combinations

### Influence of different node, process and thread configurations

Like with our GPU testing we wanted to see how much the C compiler influences the performance. For the CPU only mode we expected a greater influence of the C compiler than in the GPU mode.

Figure 4.10 shows that the best configuration for this specific input file is Clang running on 2 nodes, with 32 processes and 3 threads. The shown measurements were taken during the competition to determine which version should be used for our final run.

### 4.2.6 Results

Our preliminary tests at the competition showed us that the best combination of compiler and MPI library was Clang 3.8 with OpenMPI 1.10.2 (see Figure 4.10). However due to an oversight at the competition we used GCC combined with OpenMPI for our final run. As the judges did not release details about the runs from all the teams in the competition it is impossible for us to say if this would have made a significant difference in our judging.

As we were not using the GPU accelerated code we did not have any problems with the power limit and stayed under 1.1kW for both runs.

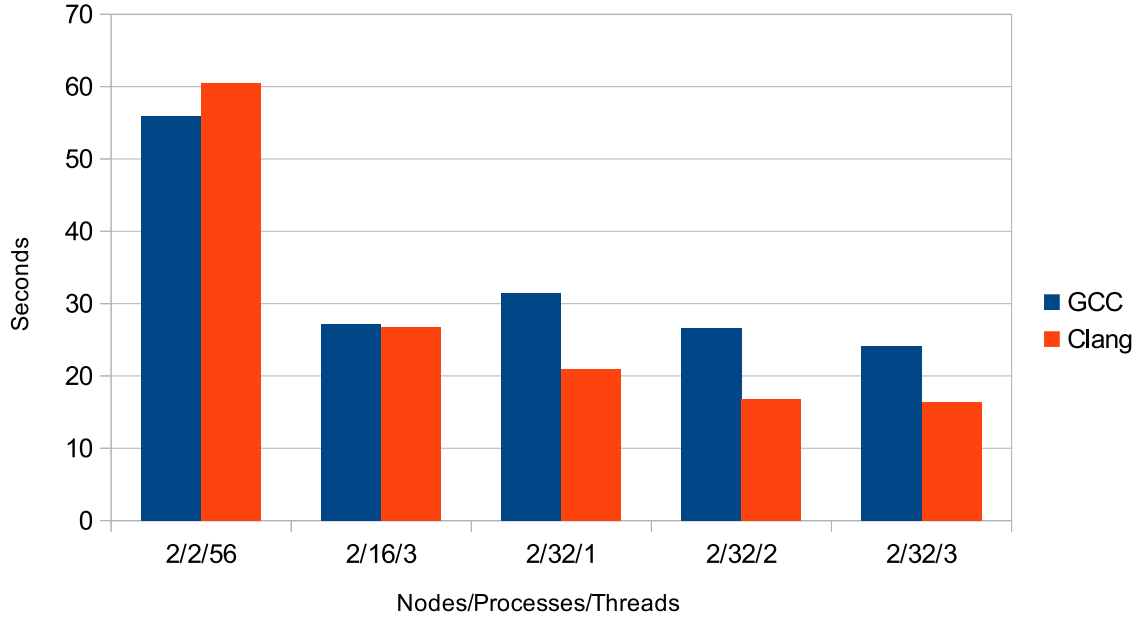


Figure 4.10: Comparison between Clang 3.8 and GCC 6, in combination with OpenMPI

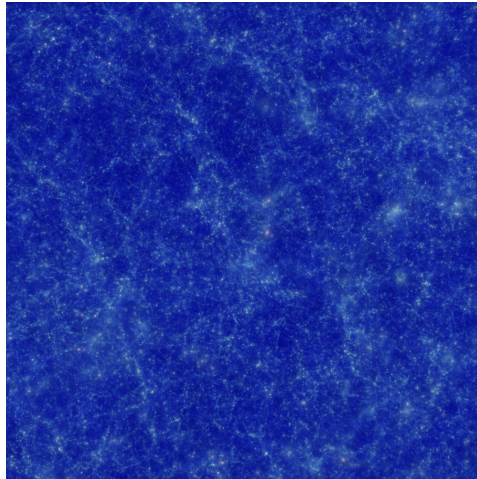


Figure 4.11: Sample image from the first competition run

Both input files generated 1000 images. The first input file was a zoom into the particle data from a given point. Our run finished with a wall clock time of *2077.6410* seconds. The second input file was an orbit around the given particle data. The second run finished with a wall clock time of *2657.6410* seconds. Animated sequences of the first and second run can be found at <https://vimeo.com/171599857> and <https://vimeo.com/171599891> respectively.

## 4.3 Graph500

*Author: Kristina Tesch*

Graph500<sup>14</sup> is a benchmark that is used to examine the suitability of supercomputing systems for data-intensive applications. It was introduced in 2010 and new Graph500 lists are published twice a year in June and November, which is similar to the well established Top500 list that ranks supercomputers according to their FLOPS performance. The metric that is used to rank systems in the Graph500 list is Traversed Edges Per Second (TEPS). The metric relates to a Breadth-First Search (BFS) that is run on a large-scale graph to obtain the performance result. A validation, which is employed after every BFS run to ensure the result is correct, returns the exact number of edges in the connected component of the starting vertex. This number is divided by the execution time of the BFS run to get the TEPS performance. In this year Graph500 was a special challenge since we had to provide our own implementation for the BFS and validation part. The parallel BFS implementation is MPI-parallelized to run on the two nodes and has GPU support. For the sake of simplicity, the validation has been implemented to be executed by a single MPI process and is parallelized with OpenMP.

### 4.3.1 Graph500 Specification

The Graph500 specification includes five parts. An execution of the benchmark begins with the **edge list generation**. This can be done in parallel, however, it is required that no locality is introduced to the graph data in this step. The edge list generator creates an edge list that contains pairs of vertex identifiers, which mark the starting point and the ending point of an edge. The total number of vertices in the graph is determined by a parameter called scale. The overall number of vertices is calculated as  $2^{scale}$ . The number of edges is controlled through the edgefactor parameter. To obtain the overall number of edges, the number of vertices is multiplied by this value. After the edge list generation, the graph data is converted into a graph representation of choice. This step is called **graph generation** in the Graph500 specification. For a distributed implementation, an adequate partitioning of the graph data should be selected at this point. The next step is to **select starting vertices** for 64 BFS runs. Those vertices should be selected in a random fashion and fulfill the condition to have at least one outgoing edge. The main part of the benchmark is the execution of **64 BFS runs** on the graph data. The time for every BFS run is measured to receive 64 performance results. A **validation** is employed after every BFS. A number of specified criteria is checked to

---

<sup>14</sup><http://www.graph500.org>



ensure the correctness of the BFS tree that is returned by every BFS execution. Finally, a **statistic on the performance** values is given.

### 4.3.2 GPU-enabled Reference Implementation

A GPU reference implementation can be downloaded from here<sup>15</sup> and is further explained in [US13]. Tests have shown that this implementation delivers quite good performance results for large graphs (scale greater than 25).

### 4.3.3 Own Implementation

The code for our own multi-node implementation of Graph500 can be found in the repository in the Graph500/Code directory. Three GPU versions are included that have been implemented with CUDA, Thrust and OpenACC as well as a CPU implementation that has been parallelized using OpenMP. The implementation is explained further in the Graph500 documentation that can be found in the repository as well.

#### Dependencies (CUDA Version)

- gcc 5.1.0

To be able to use a version of gcc > 4.9 for compiling host code, the follow lines have to be deleted in host\_config.h:

```
1 #if defined(__GNUC__)
2 #if __GNUC__ > 4 || (__GNUC__ == 4 && __GNUC_MINOR__ > 9)
3 #error -- unsupported GNU version! gcc versions later than 4.9 are
   not supported!
4 #endif /* __GNUC__ > 4 || (__GNUC__ == 4 && __GNUC_MINOR__ > 9) */
```

Listing 4.6: Lines to be deleted in order to use a gcc version > 4.9

- CUDA 7.5 and an appropriate driver
- OpenMPI v1.10.2 (CUDA-aware)

OpenMPI can be compiled with CUDA support:

```
1 module load gcc/5.1.0
2 ./configure --prefix=$(BUILD_DIR) --with-cuda=$(CUDA_ROOT_DIR)
3 make all install
```

Listing 4.7: Compile OpenMPI with CUDA support

---

<sup>15</sup><https://sites.google.com/site/tokyotechsuzumuralabeng/graph-500-challenge>

## Dependencies (CPU Version)

- gcc 5.1.0
- OpenMPI v1.10.2

## How to Build

A CMake project has been setup for every version of Graph500. The corresponding CMakeLists.txt files can be found in the particular directory. Additionally, build-scripts have been included in the repository for all versions, which load the appropriate modules before executing the cmake command. The available options are listed in the following:

Option	Description	Available
LOGGING	Enables logging.	all
NO_VALIDATION	Turns off the validation to speed up the execution time. Valid statistics are only given for EDGEFACTOR=16.	CPU, CUDA, THRUST
SCALE	Choose the Scale for the CUDA version. Scales from 16 to 27 are possible, others have to be added to src/static_settings.h.	CUDA
SETTING	Choose the launch configuration. Setting 1 is always available, other have to be added to src/static_settings.h	CUDA
AUTO_GPU_ASSIGNMENT	Turns on the automatic assignment of GPUs to MPI processes.	GPU versions
SHUFFLE	Use if a NVIDIA Kepler GPU is used. (Enables the intrinsic shuffle instructions)	CUDA
NUM_EDGES_TRAVERSED	The number of traversed edges has to be set manually if the validation is turned of for the CUDA version. The values for an EDGEFACTOR=16 can be obtained from src/settings.cpp	CUDA

Linking the GPU version requires the CUDA driver to be installed as `libcuda.so` is not found otherwise.

## How to Run

Example jobscrippts can be found in the `jobscrippts/directory`. For the CPU, the Thrust and the OpenACC versions, the scale and edgefactor are given as command line parameters for every program run. In comparison, the CUDA version has to be compiled with the required setting. The reason for this is that it simplifies the choice of the optimal launch configuration for the CUDA kernels.

At the competition we ran the CUDA version with the intrinsic Kepler instructions switched on, but turned off the automatic assignment of GPUs to MPI processes. Instead, a script was used to make only one GPU visible to each MPI process via environment variables. This was done to avoid errors related to CUDA Inter-Process Communication (IPC) failing for GPUs connected to the same node but different IOH chips.

## Results at the Competition

For the competition an edgefactor of 16 was required. If no restrictions are made, it can be assumed that a higher edgefactor will usually improve the performance for a GPU as well as a CPU implementation. The reason for this is that a higher edgefactor changes the graph's internal structure such that the CPUs' cache hierarchy can be exploited more effectively and GPUs will benefit from a higher bandwidth to the global memory through coalesced memory access.

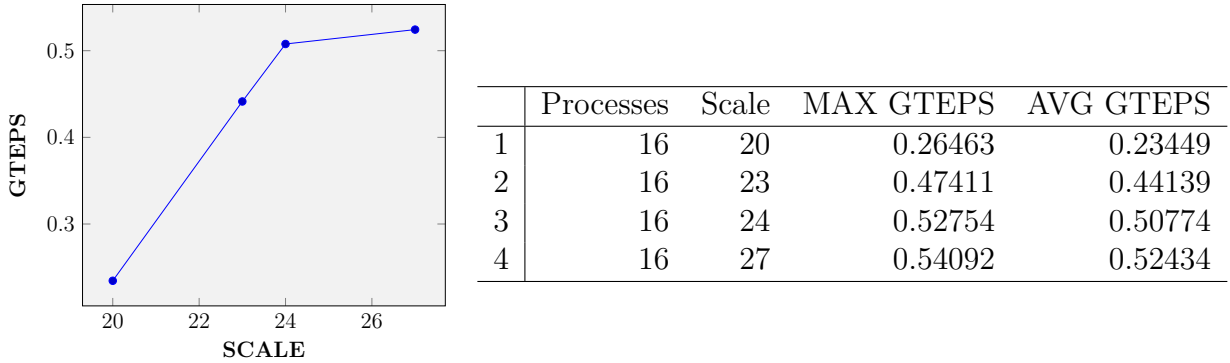


Figure 4.12: Results

As it can be seen in the table on the right side of Figure 4.12, our best result is 0.540922 GTEPS, which was achieved with scale 27.

Table 4.2 shows our results for a small weak scaling setup. We compare 4 processes with scale 18 to a setup with four times as many processes (16 processes) and a graph

	Processes	Scale	Traversed Edges	GTEPS
1	4	18	4194257	0.22587
2	16	20	16777010	0.23449

Table 4.2: Weak scaling

with four times as many nodes (scale 20). The results in the last column shows that four times as many processes can traverse more than four times as many edges per second. This corresponds to a linear/superlinear scaling.

#### 4.3.4 How to Use Vampir With a CUDA Application

Vampir is an analysis tool for parallel applications, which is useful to identify performance issues in MPI programs. An instrumented version of the application has to be executed in order to gather the event data that is visualized using Vampir afterwards. Instrumentation can be done with the Score-P framework. Listing 4.8 shows how to compile Score-P with CUDA support.

```

1 ./configure --prefix=$(BUILD_DIR) --enable-cuda --with-libcudart=$(
    CUDA_ROOT_DIR) --without-gui
2 make
3 make install

```

Listing 4.8: Compile Score-P with CUDA support

The next step is to compile the application with a Score-P wrapper. Listing 4.9 shows how the Score-P wrappers can be used together with CMake.

```

1 module load gcc/5.1.0
2 module load openmpicuda75-gcc5
3 module load scorep-openmpicuda-gcc5
4
5 SCOREP_WRAPPER=off cmake .. -DCMAKE_C_COMPILER='which scorep-gcc' -
    DCMCMAKE_CXX_COMPILER='which scorep-g++' -DMPI_C_COMPILER='which scorep-
    mpicc' -DMPI_CXX_COMPILER='which scorep-mpicxx'

```

Listing 4.9: Usage of the Score-P wrappers with CMake

After the compilation the application is executed. Environment variables are used to control the Score-P framework. Different CUDA measurement features can be enabled as visible in line 6 of Listing 4.10.

```

1 #Load appropriate modules
2

```

```

3 export SCOREP_TOTAL_MEMORY=104MB
4 export SCOREP_ENABLE_TRACING=true
5 export SCOREP_ENABLE_PROFILING=true
6 export SCOREP_CUDA_ENABLE=runtime,driver,kernel,idle,memcpy,sync
7 export SCOREP_CUDA_BUFFER=104MB
8 export SCOREP_FILTERING_FILE=$(FILTER_FILE)
9
10 # run application instrumented with Score-P

```

Listing 4.10: Score-P environment variables

### 4.3.5 NVIDIA Profiler

If only the CUDA kernels and CUDA API calls are of interest for further optimization, the NVIDIA Profiler `nvprof` provides useful information. The profiler is included in the CUDA toolkit and does not require recompiling the code. As visible in Listing 4.11, the profiling output can be stored in a separate file for each MPI processes named with the `SLURM_PROCID`. Calling `nvprof` with one of those output files provides a textual representation similar to one of `gprof`.

```

1 #gather profiling data
2 srun -n 16 $(CUDA_ROOT_DIR)/bin/nvprof -o Graph500.%q{SLURM_PROCID}.nvprof ./build/cuda/
   release/Graph500CUDA18
3
4 #view profiling data
5 $(CUDA_ROOT_DIR)/bin/nvprof --import-profile Graph500.0.nvprof

```

Listing 4.11: Usage of the Score-P wrappers with CMake

The profiling output can be viewed with the NVIDIA Visual Profiler, which provides a detailed visualization of the data. Figure 4.13 shows a part of the time line that visualizes every called function.

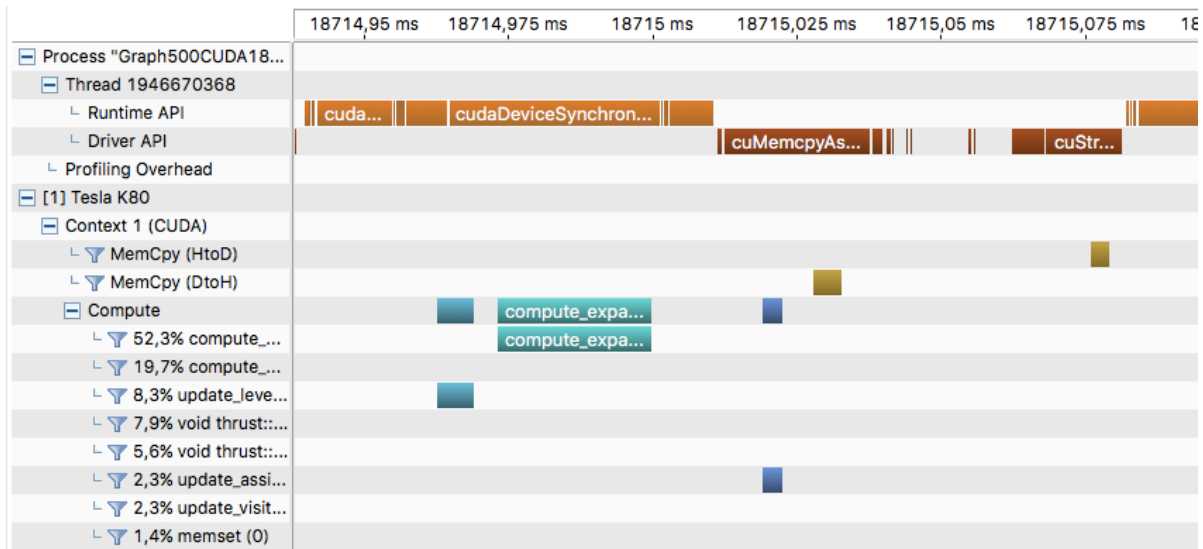


Figure 4.13: Screenshot of NVIDIA Visual Profiler

## 4.4 CloverLeaf

*Author: Jesko Regenthal, Thomas Walther*



Figure 4.14: CloverLeaf logo <sup>16</sup>

CloverLeaf<sup>17</sup> is a small application that solves the compressible Euler equations on a Cartesian grid. To achieve the maximum of optimization by the compiler all computation has been broken down into "kernels" with minimal complexity, it also sacrifices memory to achieve higher performance. Cloverleaf is available in a variety of different implementations including serial, MPI/OpenMP and CUDA.

### 4.4.1 CloverLeaf as the mystery application

CloverLeaf was revealed as an application we had to run as part of the surprise challenge. The task was to complete a CloverLeaf run in 60 minutes while using the lowest power possible. For this we were given some time to familiarize ourselves with the application and determine which implementation we want to use and if we want to remove any hardware to further reduce power consumption.

<sup>16</sup><http://uk-mac.github.io/CloverLeaf/images/logo.png>

<sup>17</sup><http://uk-mac.github.io/CloverLeaf/>

### 4.4.2 Dependencies

The dependencies of CloverLeaf are minimal, as portability was an aspect kept in mind during development. For most systems a simple make should suffice. Due to the limited time we only tested the MPI/OpenMP reference implementation, the CUDA implementation and the MPI-only implementation.

Dependencies for the tested versions are:

- MPI/OpenMP
  - MPI library
  - compiler with OpenMP support
- CUDA
  - CUDA Toolkit
- MPI
  - MPI library

### 4.4.3 How to Build

#### MPI/OpenMP

To build the MPI/OpenMP implementation we had to replace `-openmp` with `-qopenmp` in the Makefile and set the environment variables `COMPILER` to our desired compiler, `OPTIONS="-xavx2"` and `C_OPTIONS="-xavx2"`. We set the `LD_LIBRARY_PATH` to the corresponding directory in our job script.

For the CUDA version we needed to set `NV_ARCH=KEPLER`. The MPI-only version did not need any changes, all that was needed was loading the MPI Library and desired compiler with our module system.

### 4.4.4 How to Run

CloverLeaf takes no arguments, it expects a file named `clover.in` in the same directory as the binary. We wrote a simple Slurm batch script<sup>18</sup> setting the desired number of nodes, processes and threads

---

<sup>18</sup>this script can be found in the `scripts/CloverLeaf` directory

### 4.4.5 Benchmarks

In the preparation phase we compared the CUDA, MPI-only and MPI/OpenMP implementations and found out, that the reference MPI/OpenMP implementation performed best for us. In addition the GPUs were not necessary for our WRF run, so we did not have to weigh the potential performance benefits of the GPUs against the higher wattage and we were able to remove all our GPUs from our cluster before starting the measured runs.

We were given two `clover.in` files, one for a shorter run to test different compilers and one for the run we had to submit.

With the smaller file we determined that Intel in combination with `mvapich2.2b` was our best option. We did our test runs only on 1 node with 56 processes to see if this would be enough to complete the task in one hour.

In our first test run with the small input data, we used the base clock of 2.4 GHz for all 56 processes and got a result after round about 212 seconds. While the run, we got an output for every step the benchmark was computing, so we were able to calculate the remaining time to get the result. Because we had limited time, we started another test run with a bare clock speed to see how it perform. Because we only needed 212 seconds for a run 10% the size of the real run, we decided to calculate on a time of 300 seconds. The result was a scaling of 0.7 we applied to our clock speed and got the clocking of 1.68 GHz.

We started the test run and calculated with the first output. The results showed us, that we would get the final result in time and we stoped the test run, because we overextended the given time for WRF and had to hurry. Another point we included in our decision, was the option to raise the clock after starting the run, if our calculations would say, that we would not make it in time.

While the final run we observed the output and calculated the remaining time every few minutes, but it showed up, that our calculations were on the point and we would be finished in round about 54 minutes.

### 4.4.6 Results

Were able to finish our CloverLeaf run in 3201 seconds while staying under 0.6 kW. This was the best performance of all teams and we scored a 100% for this challenge.



# Bibliography

- [US13] K. Ueno and T. Suzumura. Parallel distributed breadth first search on GPU. In *20th Annual International Conference on High Performance Computing*, pages 314–323, Dec 2013.

# List of Figures

1.1	Picture of our Team members and Supervisors . . . . .	4
1.2	Our node monitoring setup . . . . .	6
1.3	Our booth (left), booths of multiple teams (right) . . . . .	6
2.1	SYS-1028GQ-TR from Supermicro <sup>19</sup> . . . . .	7
2.2	CentOS logo <sup>20</sup> . . . . .	9
2.3	Slurm logo <sup>21</sup> . . . . .	10
2.4	Spack logo <sup>22</sup> . . . . .	10
2.5	Filesystem structure of our two compute nodes . . . . .	11
2.6	GlusterFS antmascot <sup>23</sup> . . . . .	12
2.7	BeeGFS Logo <sup>24</sup> . . . . .	12
2.8	Spack logo <sup>25</sup> . . . . .	13
3.1	Benchmarking different quantities of the FERMI HPL . . . . .	20
3.2	Benchmarking different quantities of the NVIDIA HPL binary . . . . .	21
3.3	Comparison of NB and resulting GFLOPS . . . . .	25
4.1	An example visualization (test run) . . . . .	30
4.2	Comparison of compiler and MPI library combinations . . . . .	33
4.3	Benchmark of <i>icc@17.0.0</i> in combination with <i>mvapich2@2.2b</i> . . . . .	34
4.4	Comparison of generated images by CUDA (right) and CPU only (left) runs . . . . .	37
4.5	Influence of the number of used GPU Cores on the CUDA setup time . .	39
4.6	Comparison between Clang 3.8 and GCC 4.8.5 . . . . .	39
4.7	Comparison between OpenCL and CUDA run mode . . . . .	40
4.8	Comparison between OpenCL and CPU run mode . . . . .	41
4.9	Comparison compiler, MPI library combinations . . . . .	42
4.10	Comparison between Clang 3.8 and GCC 6, in combination with OpenMPI	43
4.11	Sample image from the first competition run . . . . .	43
4.12	Results . . . . .	47
4.13	Screenshot of NVIDIA Visual Profiler . . . . .	50

4.14 CloverLeaf logo <sup>26</sup> . . . . .	50
--	----

# List of Tables

4.1	Our results of day 2 . . . . .	32
4.2	Weak scaling . . . . .	48

# Listings

2.1	Template script . . . . .	14
2.2	Reproducible Slurm installation script . . . . .	15
2.3	Reproducible BeegFS installation script . . . . .	17
3.1	Setting Clockspeed of CPU and GPU . . . . .	21
3.2	Our best result which broke previous WR . . . . .	22
4.1	configuration case for a compilation . . . . .	29
4.2	How to use the install script . . . . .	35
4.3	How to use the run script . . . . .	36
4.4	Comparison between multiple K80s and a single desktop grade GTX570 .	37
4.5	Excerpt from a OpenCL log file . . . . .	40
4.6	Lines to be deleted in order to use a gcc version > 4.9 . . . . .	45
4.7	Compile OpenMPI with CUDA support . . . . .	45
4.8	Compile Score-P with CUDA support . . . . .	48
4.9	Usage of the Score-P wrappers with CMake . . . . .	48
4.10	Score-P environment variables . . . . .	48
4.11	Usage of the Score-P wrappers with CMake . . . . .	49