An execution time and energy model for an energy-aware execution of a conjugate gradient method with CPU/GPU collaboration

Jens Lang[a,*], Gudula Rünger[a]

[a]*Department of Computer Science, Chemnitz University of Technology, 09107 Chemnitz, Germany*

Abstract

The parallel preconditioned conjugate gradient method (CGM) is used in many applications of scientific computing and often has a critical impact on their performance and energy consumption. This article investigates the energy-aware execution of the CGM on multi-core CPUs and GPUs used in an adaptive FEM. Based on experiments, an application-specific execution time and energy model is developed. The model considers the execution speed of the CPU and the GPU, their electrical power, voltage and frequency scaling, the energy consumption of the memory as well as the time and energy needed for transferring the data between main memory and GPU memory. The model makes it possible to predict how to distribute the data to the processing units for achieving the most energy efficient execution: The execution might deploy the CPU only, the GPU only or both simultaneously using a dynamic and adaptive collaboration scheme. The dynamic collaboration enables an execution minimising the execution time. By measuring execution times for every FEM iteration, the data distribution is adapted automatically to changing properties, e.g. the data sizes.

Keywords: conjugate gradient method; energy awareness; energy model; execution time model; RAPL; GPU

*Corresponding author, tel.+4937153137474, fax +49371531837474
Email addresses: jens.lang@cs.tu-chemnitz.de (Jens Lang), ruenger@cs.tu-chemnitz.de (Gudula Rünger)

Preprint submitted to Elsevier

1. Introduction

For simulations in science and technology, fast and energy-efficient numerical methods are essential. Depending on the algorithm, graphics processing units (GPUs) have proven to be faster [1, 2] and more energy-efficient [3, 4, 5] than CPUs. This work investigates an energy-aware execution of a parallel preconditioned conjugate gradient method (CGM) which is used in an adaptive finite element method (FEM).

Both kinds of processing units, CPU and GPU, can execute the CGM. The possibility of moving the computation freely between the processing units provides the opportunity to choose the most energy-efficient way of execution. Depending on the characteristics of the machine, the best workload distribution may be: executing the whole workload on the CPU, executing the whole workload on the GPU, or deploying a CPU/GPU collaboration where each processing unit executes a part of the workload.

For an energy-aware workload distribution, an application-specific execution time and energy model is developed. The model is based on various experiments and considers the execution speed of the CPU and the GPU, their electrical power, voltage and frequency scaling, the energy consumption of the memory as well as the time and energy needed for transferring the data between main memory and GPU memory. For the cases where the model yields the CPU/GPU collaboration as the most energy-efficient way of execution, a dynamic and adaptive CPU/GPU collaboration scheme is exploited. This scheme minimises the execution time of the CGM by measuring a number of parameters during the execution and predicting the optimal workload distribution for the CGM of the next FEM iteration.

According to [6], "support for modelling, measurement, and analysis, and autotuning on/for heterogeneous hardware platforms" as well as "energy-efficiency adaptation" mark important milestones on the way to the next-generation computing systems, such as exascale computing. This article contributes to reaching these milestones by developing an execution time and energy model for the execution of a CGM on a heterogeneous CPU-GPU platform. The model provides the possibility to distribute the workload between CPU and GPU in an optimal way with respect to the execution time or the energy consumption. The model enables to choose between execution on CPU only, GPU only, or using CPU/GPU collaboration. For the collaboration, an online autotuning method minimising the execution time is exploited which does not only take into account characteristics of the machine and the input data, but also considers influences which are changing during the simulation, e.g. the increasing number of mesh nodes. Furthermore, the fact that AMD announced *heterogeneous queuing* [7] for its upcoming generation of advanced processing units (APUs), i.e. processing units combining CPU and GPU, shows the significance of such approaches. The principle of heterogeneous queuing is based on a central, shared queue of workload which is distributed automatically to heterogeneous processing units, i.e. the CPU or the GPU part of the APU. Partly, heterogeneous queuing does in hardware what this article proposes to implement in software.

This article will progress as follows: The application, a parallel preconditioned conjugate gradient method, is explained in Sect 2. Section 3 presents the methodology used for measuring execution times and energy consumption as well as the experiments conducted. The experiments lead to an execution time and energy model which is developed in Sect. 4. In that section, also the approaches for finding an energy-aware workload distribution are discussed. The dynamic workload distribution method is presented in Sect. 5. Section 6 presents related work and Sect. 7 concludes the article.

2. Parallel preconditioned conjugate gradient method

The scientific code considered is an adaptive finite element method (FEM) [8] which is applied to deformation problems. The FEM refines its mesh adaptively at the most critical points in contiguous iterations, i.e. it adds more elements to the mesh at the points with the largest deformation gradients. The following steps are executed consecutively until a given accuracy is reached: (I) adaptive, instead of total, mesh refinement, (II) assembly of the stiffness matrices, (III) solution of a linear system of equations with the conjugate gradient method, and (IV) error estimation for next refinement. Step (III) is the most time-consuming step of the FEM and investigated in this article.

2.1. Sequential method

The conjugate gradient method in step (III) solves the linear system of equations with the stiffness matrix A and the load vector b

$$Au = b \tag{1}$$

by iterating and testing for an approximation u_k if $Au_k \approx b$ holds. More precisely, it is tested whether the residuum $r_k = |Au_k - b|$ is below a given error bound e, i.e. $r_k < e$. If the condition is not satisfied, the next approximation u_{k+1} is computed and tested in a further iteration. The first approximation u_1 is chosen arbitrarily. Using a preconditioner for choosing u_{k+1} accelerates the convergence and hence reduces the number of iterations needed.

The routine PPCGM as implemented in [8, 9] is shown as pseudo code in Alg. 1. The routine executes the conjugate gradient method as described. PPCGM calls the routine AXMEBE to calculate the matrix-vector multiplication $y = Au$. AXMEBE does not use the full vectors u and y and the full matrix A, but performs an element-wise matrix-vector multiplication of the form:

$$y_{el} = A_{el} u_{el} \tag{2}$$

for all finite elements el of the FEM. While the size of the full vectors grows proportionally to the number of elements and may grow to some hundreds of thousands, the size of the element vectors remains constant. Depending on the number of nodes per elements and the number of degrees of freedom, the size of an element vector is between 8 and 81. The data needed for processing one element, i.e. the element data structures A_{el}, u_{el} and y_{el}, is extracted from the full data structures and converted back by dedicated functions o2el and el2o:

$$A_{el} = \mathrm{o2el}(A, el) \quad , \quad u_{el} = \mathrm{o2el}(u, el) \quad , \tag{3}$$

$$y = \sum_{el} \mathrm{el2o}(y_{el}) \quad . \tag{4}$$

Since the equations (2) to (4) are independent from each other for each element, AXMEBE can be executed in parallel, as shown in Alg. 1 (Lines 12 to 18): In Line 13, the element data structures are extracted from the full data structures according to Eq. (3). The matrix-vector multiplication of Eq. (2) is performed in Line 14, and the element data structures are converted back according to Eq. (4) in Line 16. For writing into the full vector y, which is shared between the processors, the write accesses between have to be synchronised by a mutual exclusion denoted as *critical section* in the pseudo code.

2.2. Parallel CPU/GPU execution

The parallel section in lines 12 to 18 of Alg. 1 is suitable for being executed on both, CPU or GPU. With parallel execution on the CPU and on the GPU, the high computational power of the GPU can be exploited without leaving the CPU idle in the meantime. The implementation is shown in Alg. 2 [10]. The bars and the symbols on the right-hand side indicate execution time parameters which will be needed in the following Subsect. 2.3.

For a parallel execution on p CPU cores and on the GPU, the workload W consisting of n_{el} elements to be processed has to be split up into disjoint sets, one for each processing unit (Line 3). $W_{\mathrm{cpu}k}$ denotes the workload to be processed on the kth CPU core, $1 \leq k \leq p$, and W_{gpu} and denotes the workload to be processed on the GPU. The data required on the GPU has to be transferred to the GPU memory before the execution and the result has to be transferred back afterwards. The current approximation of the load vector u and the result vector y_{gpu} have to be transferred once for every call of AXMEBE (Lines 20 and 29). The parts of the stiffness matrix A needed for creating A_{el} and the permutation vector for transforming A into A_{el} for the elements el processed on the GPU have to be transferred once per PPCGM call (lines 5 and 6).

The routine AXMEBE is executed by multiple threads. Each of the threads has a unique thread id tid, $0 \leq tid \leq p$. The first thread with $tid = 0$ (see Line 19) is responsible for transferring u and y_{gpu} to the GPU memory as well

```
1  PPCGM(A, b)
2  begin
3      repeat
4          call preconditioner
5          calculate next u
6          y := AXMEBE(A, u)
7      until y ≈ b
8      return u
9  end
10 AXMEBE(A, u)
11 begin
12     for each el do in parallel
13         A_el := o2el(A, el),   u_el := o2el(u, el)
14         y_el := A_el u_el
15         begin critical section
16             y := y + el2o(y_el)
17         end critical section
18     next
19     return y
20 end
```

Algorithm 1: Pseudo code of PPCGM and AXMEBE

as for launching the GPU kernels. Further p threads perform the computation on the CPU cores. At the end of AXMEBE, the two result vectors y_{gpu} and y_{gpu} are added and returned to PPCGM.

The code used here is based on an existing shared-memory parallel implementation of the adaptive FEM in OpenMP [9]. The GPU version of AXMEBE has been developed in CUDA. An implementation using CUBLAS has proven to be inefficient, see [10].

2.3. Distribution-dependent parameters

For the prediction of the execution time and the energy consumption, a model is developed. In order to find an optimal distribution between the processing units, the model considers all sections of the algorithm that depend on the distribution between CPU and GPU. They are marked by the bold vertical lines in Alg. 2. The lines are annotated by symbols for the execution times of the respective sections: t_{copy} denotes the time for copying the required data to the GPU memory, t_{gpu} denotes the time needed for processing one element on the GPU and t_{cpu} denotes the time needed for processing one element on the CPU. The energy consumption of these three sections is denoted by E_{copy}, E_{gpu} and E_{gpu}, respectively.

In Sect. 3, laws describing the behaviour of the quantities with respect to parameters such as the number of elements to be processed or the clock frequency of the CPU are investigated. Based on the investigations, an application-specific model is set up in Sect. 4.

3. Experiments

For an accurate prediction, it is necessary to find laws describing the behaviour of the values

- t_{cpu} and E_{cpu} for processing the workload on the CPU,
- E_{dram} for the energy needed by the DRAM during the processing on the CPU,
- t_{gpu} and E_{gpu} for processing the workload on the GPU, as well as
- t_{copy} and E_{copy} for transferring the data from the CPU to the GPU.

These laws may consider the number of elements processed on the respective processing units, further characteristics of the data as well as hardware properties. Based on the preliminary work in [11], experiments conducted for finding the laws are described in this section.

Not only laws but also methods for measuring the values have to be investigated. In order to verify the measurement methods, results obtained in the experiments are compared to values obtained from other sources. All measurement methods use only software interfaces; no additional measurement hardware is used.

```
1  PPCGM(A, b)
2  begin
3      distribute all el ∈ W to W_gpu and W_cpuk, 1 ≤ k ≤ p such that |W_gpu| = n_el,gpu and |W_cpuk| = n_el,cpu
4      for each el ∈ W_gpu do
5          copy part of A for el to GPU memory                          | t_copy
6          copy permutation data to GPU memory                          |
7      next
8      repeat
9          call preconditioner
10         calculate next u
11         y := AXMEBE_CPU_GPU(A, u)
12     until y ≈ b
13     return u
14 end
15 AXMEBE_CPU_GPU(A, u)
16 begin
17     begin parallel
18         tid := ID of current thread, 0 ≤ tid ≤ p
19         if tid == 0 then
20             transfer u to GPU memory                                 |
21             for each el ∈ W_gpu do (on GPU)                          |
22                 A_el := o2el(A, el)                                  |
23                 u_el := o2el(u, el)                                  |
24                 y_el := A_el u_el                                    | t_gpu
25                 begin critical section                               |
26                     y_gpu := y_gpu + el2o(y_el)                      |
27                 end critical section                                 |
28             next                                                     |
29             transfer y_gpu to main memory
30         else
31             for each el ∈ W_cputid do                                |
32                 A_el := o2el(A, el)                                  |
33                 u_el := o2el(u, el)                                  |
34                 y_el := A_el u_el                                    | t_cpu
35                 begin critical section                               |
36                     y_cpu := y_cpu + el2o(y_el)                      |
37                 end critical section                                 |
38             next                                                     |
39         end
40     end parallel
41     return y_cpu + y_gpu
42 end
```

Algorithm 2: Pseudo code of PPCGM and AXMEBE for CPU and GPU

machine	*Westmere*	*Sandybridge*
CPU model	**Intel Xeon X5650**	**Intel Xeon E5-2650**
CPU clock rate	2.67 GHz	2.0 GHz
CPUs, cores, threads	2 × 6 × 2 = 24	2 × 4 × 2 = 16
CPU base peak performance	64 Gflop/s	128 Gflop/s
CPU thermal design power	95 W	95 W
RAM model	Samsung M393B5670EH1-CH9	Samsung M393B1K70DH0
DIMMs, size per DIMM	6 × 2 GiB = 12 GiB	4 × 8 GiB = 32 GiB
RAM speed	1333 MHz	1600 MHz
NUMA nodes	2	2
GPU model	**Nvidia GeForce GTX 570 HD**	**Nvidia Tesla C2075**
GPU memory	1.2 GiB	6.0 GiB
GPU memory speed	1.9 GHz	1.5 GHz
CUDA cores	480	448
CUDA core frequency	1.46 GHz	1.15 GHz
GPU peak performance	176 Gflop/s	515 Gflop/s
Memory bandwidth	152 GB/s	208 GB/s
GPU thermal design power	219 W	215 W
Operating system	Linux kernel 3.2.21	Linux kernel 3.2.46
GPU driver version	304.48	304.88
CUDA version	4.2	5.0

Table 1: Specifications of the machines used for the experiments [12, 13, 14, 15, 16, 17]

3.1. Experimental setup

The experiments were conducted on two different machines, viz *Westmere* and *Sandybridge*. Their characteristics are given in detail in Tab. 1. The machine *Westmere* is a 2×6-core Xeon machine with a GeForce GTX 570 HD GPU. The machine *Sandybridge* is a 2×4-core Xeon machine with a Tesla C2075 GPU. As only the *Sandybridge* machine has software-readable energy and power meters, the energy experiments have been conducted on this machine. The experiments concerning the execution time and the dynamic workload distribution between CPU and GPU have been conducted on both machines.

Two different input data files, the test objects *drill hole*, representing a cuboid with a drill hole, and *crankshaft*, have been used for the experiments. The objects are shown in Fig. 1 and have been taken from the library provided with [8]. The input parameters are chosen such that each finite element of the test objects consists of 27 nodes, each having 3 degrees of freedom, which results in an element data structure size of 81.

The measurements are performed executing the FEM with different test objects. Hence, all effects, such as branch mispredictions, cache misses, etc., occurring during production runs are included in the measurements. The functions reading the current time or energy status are called immediately before and after the code sections investigated. If both, time and energy are measured, the energy measurement is started after and stopped before the time measurement. During the execution of the FEM, no other applications are run on the machines in order to avoid distortions in the measurement.

In some charts showing measurement results, regression curves for the values are given. These regression curves are obtained using the Levenberg-Marquardt least-squares method.

3.2. Measuring method

For the CPU and for the GPU, different measurement methods are necessary as they have different interfaces, especially for the energy consumption. The methods are described in this section.

3.2.1. CPU

The execution times are measured by reading the high-resolution system time using the function `PAPIF_get_virt_usec` of the PAPI library [18] which uses the `clock_gettime` system call. For mea-

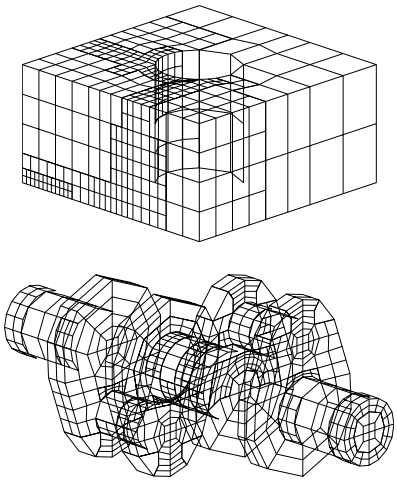

Figure 1: Test objects *drill hole* (top) and *crankshaft* (bottom), each after 8 refinement steps, from [8].

suring the energy consumption of the CPU, the machine-specific registers (MSRs) of "Running Average Power Limiting" (RAPL) [19] are used.

RAPL has been introduced by Intel with the CPU architecture code-named *Sandy Bridge*. Among the RAPL registers are "energy status" registers which provide an energy metering interface [20, vol. 3B, ch. 14-28]. On the CPU present in the machine *Sandybridge*, there are three different energy metering registers which measure the following values:

- `MSR_PKG_ENERGY_STATUS` for the energy consumption of the whole CPU package,
- `MSR_PP0_ENERGY_STATUS` for the energy consumption of processor cores including their caches [21], except the last-level cache [22],
- `MSR_DRAM_ENERGY_STATUS` for the memory modules on this package.

Each register is updated roughly every 1 ms and its value contains "the total amount of energy consumed" since the last time the register was cleared [20, vol. 3B, ch. 14-28]. In this article, the register `MSR_PKG_ENERGY_STATUS` is used for measuring the computation energy as it includes everything needed for computation on the CPU, i.e. including caches, un-core energy, etc. The register `MSR_DRAM_ENERGY_STATUS` is used for measuring the energy consumption of the memory. Other works have shown that the estimation of the energy consumption by RAPL matches the actual energy consumption quite well [23, 24, 25].

The register values are read using the interface provided by the *msr* kernel module. This module was adapted to ignore the capability `CAP_SYS_RAWIO` [26, ch. 39] normally required by executables reading MSRs [11].

3.2.2. GPU

The execution times on the GPU are measured by the CPU in the same way as described above. The time measurement facilities integrated in the GPU are not used as the objective of this work is not optimising the GPU code but an evaluation of the GPU execution time from the CPU perspective.

For measuring the energy consumption of the GPU, its integrated power meter is used. The principal purpose of the power meter is to ensure that the GPU does not consume more power than 225 W as specified in the PCI Express standard. The current value of the power can be accessed via the function `nvmlDeviceGetPowerUsage` of the Nvidia Management Library (NVML) [27]. The value comprises the power consumption of the whole GPU being drawn from both, the PCI socket and the additional power supply.

As the on-board power meter updates its value only every 20 ms, the method presented in [28] is used to obtain a power profile of the GPU routines with a higher temporal resolution. This method executes the routine

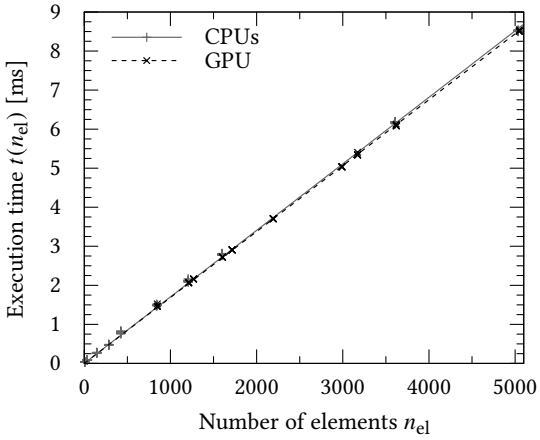

Figure 2: Execution time t of AXMEBE on the CPUs and on the GPU

to be evaluated a large number of times (50 to 100), always with the same parameters. The function is called at random phases of the sensor update interval so that the power consumption is measured at different times during the execution. The integration over all values obtained yields the energy consumption of the routine on the GPU.

Due the dependency of the power from the temperature of the circuitry [29, 30], it is necessary to ensure that the temperature of the GPU is always in the same range. A value of 54 °C has been chosen for this work. If the temperature is lower, then the GPU is heated by executing some workload. If it is higher, the GPU is left idle for some moments in order to give it the possibility to cool down. For measuring the temperature, the internal thermometer accessible via the function `nvmlDeviceGetTemperature` from the NVML is used. A temperature of 54 °C results in a static power consumption of 77 . . . 79 W for the GPU in short-idle mode.

3.3. Execution time and energy depending on the number of elements

At first, the execution time and energy consumption of AXMEBE are investigated. Figure 2 shows the execution time of AXMEBE on *Sandybridge* depending on the number of finite elements processed. Two test objects have been used in different levels of refinement. Each point in the chart represents one experiment with a certain number of elements n_{el}. For each number of elements, the experiment has been conducted 5 times. The regression lines are of the form $t_{cpu}(n_{el}) = {v_{ex}}^{-1} n_{el}$, where v_{ex} denotes the *execution speed* in terms of "elements processed per time unit". The results of the experiment show that for both processing units, CPU and GPU, the execution time t_{cpu} and t_{gpu} of AXMEBE is perfectly proportional to n_{el}:

$$t \sim n_{el} \quad . \tag{5}$$

The proportionality factors, i.e. the execution speeds, are

$$v_{ex,cpu} = 587 \, \frac{\text{el}}{\text{ms}} \quad \text{and} \quad v_{ex,gpu} = 592 \, \frac{\text{el}}{\text{ms}} \tag{6}$$

for CPU and the GPU, respectively. The CPU cores of *Sandybridge* process roughly the same number of elements in a given amount of time as the GPU. The execution time for one element on the GPU is $t_{el,gpu} = 1.69$ µs. The parallel execution time for one element on the CPU is $t_{el,cpu,\parallel} = 1.70$ µs yielding a per-element execution time of $t_{el,cpu} = 27.2$ µs as 16 threads process the elements in parallel.

Figure 3 shows the results of the energy measurement for the same experiment. As the machine contains two CPUs, the values obtained for the MSRs MSR_PKG_ENERGY_STATUS for both CPUs have been summed up

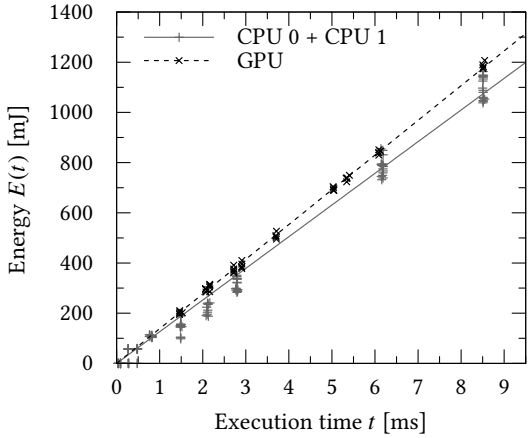

Figure 3: Energy consumption E of the CPU and the GPU for executing AXMEBE depending on its execution time t

to the total value. The results show that the energy consumption E is roughly proportional to the execution time t:

$$E \sim t \quad . \tag{7}$$

Here, the slope of the regression lines indicates the power consumption of the processing units: $E(t) = Pt$. The power values have been obtained as follows: $P_{\text{ex,cpu}} = 126$ W for the CPUs and $P_{\text{ex,gpu}} = 138$ W for the GPU. The values correspond well to the thermal design powers of 95 W per CPU package [15] and 215 W for the GPU [31]: Both kinds of processing units need roughly two thirds their maximum power.

From the relations in formulae (5) and (7), the relation

$$E \sim n_{\text{el}} \tag{8}$$

can be concluded, i.e. the amount of energy consumed is proportional to the number of elements processed.

3.4. Frequency scaling

In order to reduce their power consumption, modern CPUs can be set into operational modes with lower clock frequency and lower voltage. For each mode, a *P-state* with a distinct frequency is defined. This technique is known as *dynamic voltage and frequency scaling* [32] and called *PowerNow!* or *Cool'n'Quiet* in AMD processors or *SpeedStep* in Intel processors. In this section, it is investigated whether setting a power-saving P-state for the operation of the CPU allows the execution of the CGM in a more energy-efficient way.

The Intel Xeon E5-2650 of the machine *Sandybridge* offers P-states with the following clock frequencies: 1.2 GHz to 2.0 GHz in intervals of 0.1 GHz, and 2.001 GHz, which is the *Turbo Boost* mode [33]. *Turbo Boost* runs the CPU cores with a higher than the nominal clock rate as long as certain conditions are met, including the current power consumption and the processor temperature do not exceed the manufacturer-specified values [34].

For the experiments, the CPU cores have been set to each of the available P-states consecutively. AXMEBE has been executed and the execution time as well as the energy consumption have been measured for different numbers of elements. The chart in Fig. 4 shows the behaviour of the power consumption P and the execution speed v_{ex} depending on the CPU clock frequency f. The two vertical axes have been scaled such that the power and the execution speed are drawn on top of each other for the last P-state. The measurement results have been approximated by the curves

$$P(f) = af^3 + bf + c \quad \text{and} \quad v_{\text{ex}}(f) = df + e \tag{9}$$

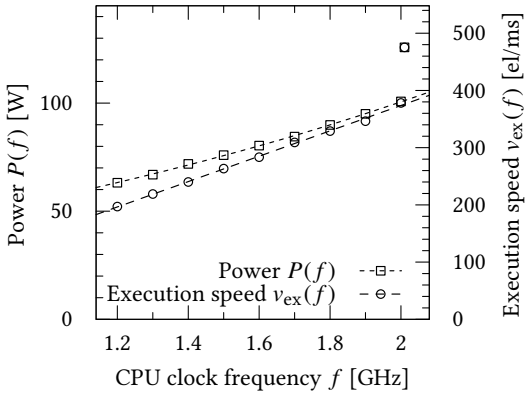

Figure 4: CPU power consumption P and execution speed v_{ex} for performing the CGM with different CPU clock frequencies f

in the range of 1.2 . . . 2.0 GHz. The *Turbo Boost* mode with 2.001 GHz has not been considered for the regression analysis as its actual frequency is defined internally and does not correspond to the clock frequency assigned to the P-state.

That the power consumption is best represented by the given cubic equation is motivated by the equation

$$P = CV^2 f + VI_{\text{leak}} \quad , \tag{10}$$

which is generally used for modelling the power consumption of CMOS circuits [35]. In this equation, V represents the voltage, C the capacitance of the transistors and wires, f the frequency and I_{leak} the leakage current. The frequency f depends linearly on the voltage V [36]. Therefore, the first summand in eq. (10) is proportional to f^3 and the second is proportional to f. For reference, the values of the parameters as obtained by the regression are: $a = 2.2 \pm 0.4$, $b = 29 \pm 4$ and $c = 24 \pm 4$.

The linear relation of execution speed and clock frequency results from the compute-boundness of the AXMEBE code. The number of instructions in the routine is fix and the duration of their execution is a fix number of clock cycles, hence the execution time of AXMEBE is roughly proportional to f^{-1}. The parameters found by the regression are $d = 222 \pm 4$ and $e = -70 \pm 6$, which means that 222 elements per millisecond times the clock frequency are processed, but 70 elements have to be subtracted. The constant offset might result from code sections needing the same execution time as 70 elements but whose execution times are invariant with respect to the clock frequency. Such code sections might be memory-accesses in AXMEBE or interrupt service routines triggered from the exterior.

The charts in Fig. 4 clearly indicate that the execution speed is increasing faster than the power consumption and hence the P-states with higher frequencies allow a more energy-efficient operation.

3.5. Per-element energy

Figure 5 shows the energy needed to process one element, the per-element energy, of the CPU and the GPU. The upper chart shows its dependency from the CPU clock speed. The results have been obtained by dividing the power values P by the execution speed values v_{ex} from Fig. 4:

$$E_{\text{el,cpu}}(f) = \frac{P(f)}{v_{\text{ex}}(f)} \quad . \tag{11}$$

The curve shown in this chart is the ratio of the curves in Fig. 4. The results show that higher clock frequencies allow a more energy-efficient operation: With a frequency of 2.0 GHz, the per-element energy $E_{\text{el,cpu}}$ is

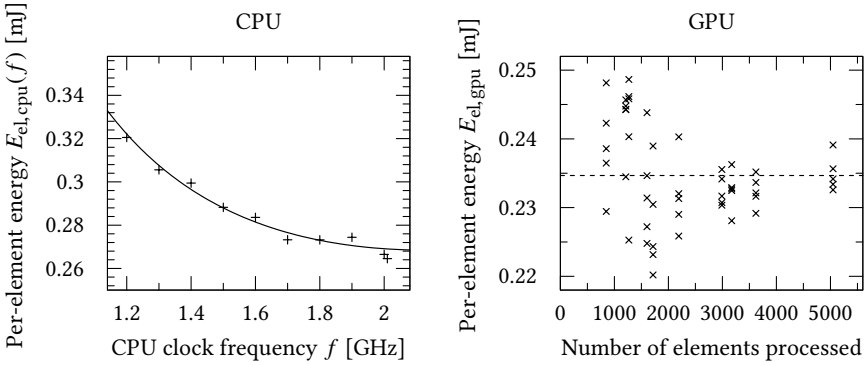

Figure 5: Energy consumption of the CPU and the GPU for processing one element

approximately 267 µJ, compared to 321 µJ with a frequency of 1.2 GHz. In the *Turbo Boost* mode, the energy is 265 µJ.

The per-element energy for the GPU $E_{\mathrm{el,gpu}}$ is shown in the lower chart in Fig. 5. Each point results from one experiment with a specific number of elements. A clear dependency from the total number of elements is not visible, $E_{\mathrm{el,gpu}}$ ranges from 220 µJ to 250 µJ. The mean of the values is 235 µJ. The results indicate that the GPU processes the workload of the CGM in a slightly more energy-efficient way than the CPU. As shown in [11], the values obtained correspond well to a value obtained theoretically by counting the operations and multiplying by an assumed energy consumption per operation found in the literature.

3.6. DRAM energy

Besides the energy needed by the processing unit for computation, the energy needed by the memory for writing, storing and retrieving the data has to be considered. For the GPU, the value measured by the NVML covers both. For the CPU, the energy needed by the cores and by the memory is measured separately and can be read from different MSRs. The energy values of the DRAMs of both CPU packages are summed up to the total memory energy E_{dram}.

Figure 6 shows the energy consumption of the DRAMs when executing AXMEBE on the CPU. 16 threads have been run on both CPUs which were in the *Turbo Boost* P-state. The chart shows that the DRAM energy E_{dram} is proportional to the execution time of AXMEBE on the CPU t_{cpu}, i.e.:

$$E_{\mathrm{dram}} \sim t_{\mathrm{cpu}} \quad . \tag{12}$$

The power of the DRAM as obtained from the regression is $P_{\mathrm{dram}} = 7.8$ W. The DRAM consists of 4 DIMMs. According to the data sheet [37], the minimum current is IDD6, called "self refresh current", and the maximum current is IDD7, called "operating bank interleave read current" [38]. The exact portions of time each current is drawn are unknown for the specific application. However, the current should always remain in the range of IDD6 to IDD7, namely 390 mA $\leq I_{\mathrm{DD}} \leq$ 3216 mA for one DIMM. With a voltage of $V = 1.35$ V and $P = VI$, the power range is roughly 2.1 W $\leq P_{\mathrm{dram}} \leq$ 17 W for 4 DIMMs. The measured value is well within the range obtained from the data sheet.

The per-element energy $E_{\mathrm{el,dram}}$ has been measured for all available CPU clock frequencies f and is shown in Fig. 7: With increasing CPU frequency, the per-element energy needed by the memory decreases. The curve indicates that the laws found for $E_{\mathrm{el,cpu}}$ in Sect. 3.4 do also apply for $E_{\mathrm{el,dram}}$. The lowest energy is needed in the *Turbo Boost* mode, $E_{\mathrm{el,dram}} = 20.5$ µJ, and the highest energy is needed with a frequency of 1.2 GHz, $E_{\mathrm{el,dram}} = 40.5$ µJ.

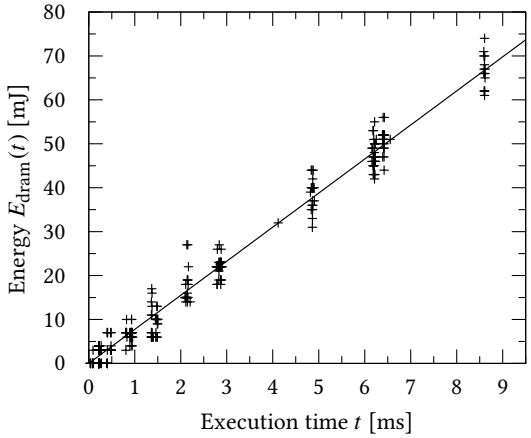

Figure 6: Energy consumption of the DRAM memory when executing AXMEBE on the CPU

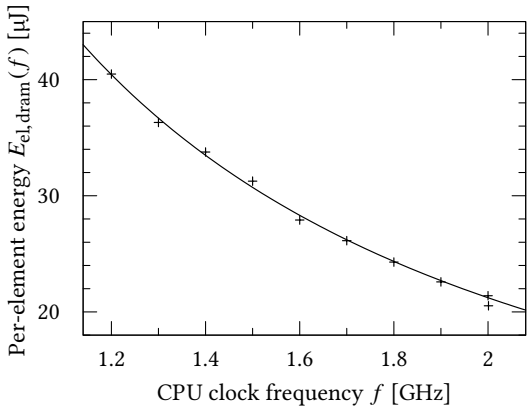

Figure 7: Per-element energy for the DRAM memory depending on the frequency

Figure 8 shows the idle power $P_{\mathrm{idle,dram}}$ of the DRAM. The CPUs have been set into sleep mode for different time intervals using `nanosleep`. During the interval, the DRAM was idle also. Each of the small points (+) represents one measurement result of the DRAM idle power, each of the large points (×) is the median of the measurements for one time interval. The overall median is represented by the horizontal line and is taken for the value $P_{\mathrm{idle,dram}} = 1.3$ W. The value for $P_{\mathrm{idle,dram}}$ is a little below the minimum power as obtained from the data sheet of 2.1 W, but an acceptable approximation.

3.7. Data transfer

In this section, the data transfer in the routine PPCGM (lines 5 and 6 in Alg. 2) is investigated. The data transferred to the GPU memory remains constant during all AXMEBE calls of one FEM refinement step. The data transfer needed for each individual AXMEBE call is already included in the values measured in Sect. 3.5.

For the transferring data from the host to the device memory and vice versa, CUDA offers the function `cudaMemcpy`. In a first experiment, the transfer rate for data stored consecutively in the memory has been determined. For amounts of data from 1 to 40 MiB, a transfer rate from the host to the device memory of 3.2 GiB/s and from the device to the host memory a transfer rate of 2.9 GiB/s has been determined [11].

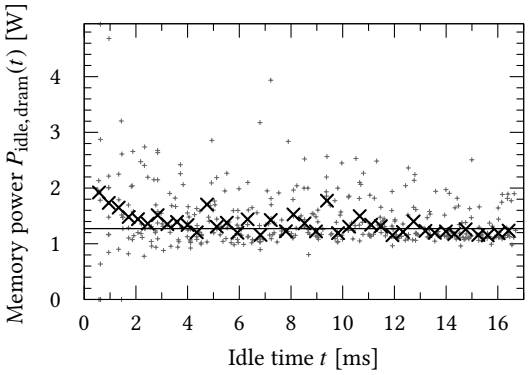

Figure 8: Idle power $P_{\text{idle,dram}}$ of the DRAM

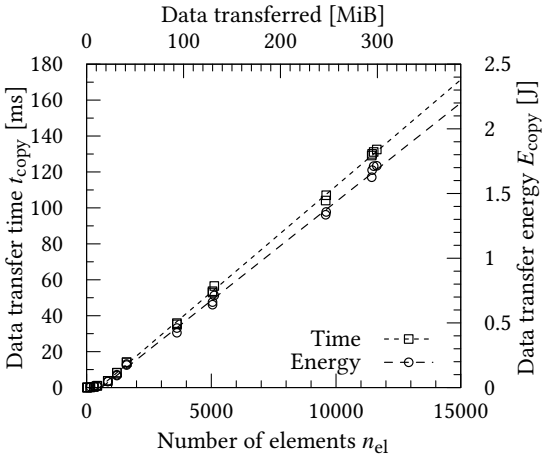

Figure 9: Time and DRAM energy consumption measured for the data transfer to the GPU memory

The energy consumption of consecutive data transfers has been investigated in [11]. However, in the application, the data structures that have to be transferred are not stored in the main memory consecutively. Consequently, the experiment is conducted within the application in the present work. From the measurements, the time t_{copy} and the energy E_{copy} needed to transfer the data to the GPU are determined. Figure 9 shows the results: Both, time and energy, exhibit linear behaviour. The lines intersect the abscissa at 500 . . . 600 elements, which corresponds to roughly 12 MiB of data. Transferring below 12 MiB of data does not cost any time or energy, possibly due to caching or similar effects.

The values shown in Fig. 9 are approximated by the following lines:

$$E_{\text{copy}}(n_{\text{el}}) = 153\ \frac{\mu\text{J}}{\text{el}} \cdot (n_{\text{el}} - 592\ \text{el}) \tag{13}$$

$$t_{\text{copy}}(n_{\text{el}}) = 11.8\ \frac{\mu\text{s}}{\text{el}} \cdot (n_{\text{el}} - 512\ \text{el}) \tag{14}$$

The data is transferred separately for each element and for each data structure. The data-transfer rate which can be derived from the chart is roughly 2.1 GiB/s. It is not surprising that this value is lower than the value of 3.2 GiB/s for the transfer of data stored in the memory consecutively.

Furthermore, a power consumption of roughly 13 W can be estimated by dividing the slope of the energy 153 µJ/el by the slope of the time 11.8 µs/el. For the data transfer, the DRAM power is higher than for the execution of AXMEBE as for the former there are more memory accesses per time.

The power consumption of the GPU has been determined to be 83.2 W during the data transfer. Writing to the device memory needs 25.7 mJ/MiB in total and reading needs 29.1 mJ/MiB, which is 3.06 nJ/bit and 3.47 nJ/bit, respectively. It is important to note that it is not possible to isolate the power consumption of the memory of the GPU, so the values in this section also comprise the power needed by the GPU processor.

Given that an element consists of 81 entries, 27 kB of data have to be transferred to the GPU per PPCGM call for each element processed on the GPU. Multiplying this value by the energy consumption of the GPU for writing to the GPU memory, which is 24 nJ/B, results in an energy consumption of 661 µJ per element.

3.8. Throttling down CPUs

As the GPU allows a more energy-efficient execution of AXMEBE than the CPU, it might be considered processing all elements on the GPU and throttling down the CPUs meanwhile. Experiments conducted showed that the throttling down does not work to the full extent: The first CPU package, CPU 0, does always draw the full amount of power, even when it is only waiting for the GPU to finish. Therefore, it is reasonable that this CPU performs some computation during the waiting period. However, the second package, CPU 1, is throttled down automatically if there is no thread running on this CPU. Running the 8 threads on the cores belonging to CPU 0 and no thread on CPU 1 is ensured with gcc's OpenMP by setting the environment variable `OMP_NUM_THREADS` to `8` and `GOMP_CPU_AFFINITY` to `0-7`.

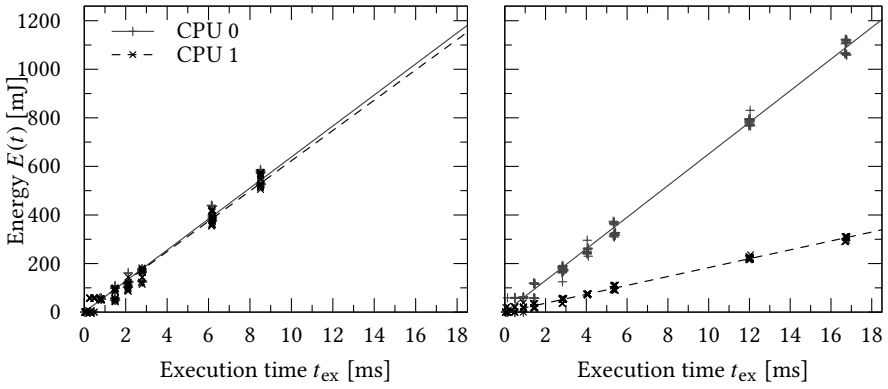

Figure 10: Energy consumption of the CPU (highest frequency) when executing AXMEBE on both CPUs with 16 threads (left) and only on CPU 0 with 8 threads while throttling down CPU 1 (right)

The left-hand chart in Fig. 10 shows the energy consumption of the CPUs when executing 16 threads: Both CPUs are working under full load and have a similar power consumption: $P_{\mathrm{CPU0}} \approx P_{\mathrm{CPU1}} \approx 63$ W. The right-hand chart shows that, in contrast, the CPU 1 has a lower power when only 8 threads, which are all assigned to CPU 0, are running in total. The CPUs have a power of $P_{\mathrm{CPU0}} = 65$ W and $P_{\mathrm{CPU1}} = 18$ W. Considering, of course, that one CPU with 8 threads needs double the time compared to 2 CPUs and only power, but not energy is saved.

3.9. Discussion

The experiments showed that the execution times t_{cpu} and t_{gpu} are proportional to the numbers of elements processed on the respective processing unit. The same is true for the energy consumptions E_{cpu} and E_{gpu}. For the data transfer, a linear relation could be found with an offset of roughly 500 elements. For simplicity, one might

consider neglecting this offset and use the proportionalities

$$t_{\text{copy}} \sim n_{\text{el}} \tag{15}$$

$$E_{\text{copy}} \sim n_{\text{el}} \tag{16}$$

instead.

For the machine *Sandybridge*, the time needed for processing one element on the CPU in parallel is $t_{\text{el,cpu,}\parallel} = 1.70\,\mu\text{s}$. For the GPU, the time is $t_{\text{el,gpu}} = 1.69\,\mu\text{s}$. The energy needed for processing one element is $E_{\text{el,gpu}} = 235\,\mu\text{J}$ for the GPU, and the lowest possible energy for the CPU is $E^*_{\text{el,cpu}} = 286\,\mu\text{J}$ including the DRAM energy. For transferring the workload to the GPU, an energy of $E_{\text{el,copy}} = 814\,\mu\text{J}$ is needed, see Sect. 3.7. The energy consumption of the CPU is neglected as there is the CUDA function `cudaMemcpyAsync` which performs the data transfer asynchronously with respect to the CPU, so that no additional CPU time is needed. Hence, the execution on the GPU is more energy-efficient than the execution on the CPU if there are more than 15 CGM iterations, i.e. calls of AXMEBE per PPCGM call.

For processing the workload, there exist three choices for its distribution between the CPU and the GPU on the machine *Sandybridge*:

1. All workload is processed on the CPUs.
2. All workload is processed on the GPU.
3. The workload is balanced between CPU and GPU.

For the third choice, a balancing method which generates that the CPUs and the GPU finish their workload in an equal amount of time can be used. Such a balancing avoids idle times in which energy is consumed but no output is produced. Also, a variant throttling down one CPU and balancing the workload between the GPU and the other CPU is worth to be considered.

4. Model

In this section, a model predicting the execution time and the energy consumption is developed extending and combining existing, preliminary models for execution time [10] and energy [11]. The basic model derived from the pseudo code in Alg. 2 is refined using the experimental results in Sect. 3. Furthermore, a method minimising the energy consumption by adapting the workload distribution is discussed.

4.1. Execution time

The execution time of the CGM consists of two main parts: the time needed for the data transfer in lines 4 and 6 of Alg. 2 and the time needed for executing the loop in lines 8 to 12. The execution time of the loop is the number l of loop iterations multiplied by the maximum of the workload execution times on the processing units:

$$t_{\text{cgm}} = t_{\text{copy}} + l \cdot \max\left(t_{\text{cpu}}, t_{\text{gpu}}\right) \quad . \tag{17}$$

The processing unit which finishes its workload first waits for the other to finish too. The code sections whose execution time does not depend on the workload distribution, e.g. lines 9 and 10 in Alg. 2 are neglected.

The values t_{cpu}, t_{gpu} and t_{copy} can be expressed by the respective values $t_{\text{el,cpu}}$, $t_{\text{el,gpu}}$ and $t_{\text{el,copy}}$ for one element, due to the proportionality found in Sect. 3, i.e. formulae (5) and (15):

$$t_{\text{cpu}} = n_{\text{el,cpu}} \cdot t_{\text{el,cpu}} \tag{18}$$

$$t_{\text{gpu}} = n_{\text{el,gpu}} \cdot t_{\text{el,gpu}} \tag{19}$$

$$t_{\text{copy}} = n_{\text{el,gpu}} \cdot t_{\text{el,copy}} \quad . \tag{20}$$

The value $n_{\text{el,cpu}}$ denotes the number of elements processed on one CPU core, $n_{\text{el,gpu}}$ denotes the number of elements processed on the GPU. The sum of the elements processed on p CPU cores and on the GPU is the total

number of elements:

$$n_{\text{el}} = p \cdot n_{\text{el,cpu}} + n_{\text{el,gpu}} \quad . \tag{21}$$

Inserting t_{cpu}, t_{gpu} and t_{copy} from the equations (18) to (20) into equation (17) yields

$$t_{\text{cgm}} = n_{\text{el,gpu}} \cdot t_{\text{el,copy}} + l \left(\max \left(n_{\text{el,cpu}} \cdot t_{\text{el,cpu}}, n_{\text{el,gpu}} \cdot t_{\text{el,gpu}} \right) \right) \quad . \tag{22}$$

For the decision whether some elements are processed on the GPU, the following inequality is considered:

$$t_{\text{el,copy}} < l \cdot \frac{t_{\text{el,cpu}}}{p} \quad . \tag{23}$$

The GPU is not used for processing elements if the inequality (23) does not hold true: If the transfer of one element's data to the GPU memory consumes more time than processing it on the CPU, then all elements are processed on the CPU, i.e. $n_{\text{el,gpu}} = 0$.

The first parameter of the max term in equation (22) is monotonically decreasing and the second term monotonically increasing with increasing $n_{\text{el,gpu}}$. Assuming that the inequality (23) is given, the execution time in equation (22) becomes minimal if

$$n_{\text{el,cpu}} \cdot t_{\text{el,cpu}} = n_{\text{el,gpu}} \cdot t_{\text{el,gpu}} \quad . \tag{24}$$

Alternatively, $n_{\text{el,cpu}}$ and $n_{\text{el,gpu}}$ can be expressed by using the rate r of the number elements processed on the GPU $n_{\text{el,gpu}}$ to the total number of elements n_{el}, $0 \leq r \leq 1$, i.e.

$$n_{\text{el,gpu}} = r n_{\text{el}} \quad , \quad n_{\text{el,cpu}} = \frac{1 - r}{p} n_{\text{el}} \quad . \tag{25}$$

Inserting the equations (25) into equation (24) yields the following expression for r:

$$r = \left(1 + \frac{t_{\text{el,gpu}}}{t_{\text{el,cpu}}} p \right)^{-1} \quad . \tag{26}$$

Of all possible distributions of the workload to the CPU and the GPU, the distribution with r chosen according to equation (26) has the lowest execution time t_{cgm} for the routine PPCGM.

4.2. Energy

The energy consumption of PPCGM is the sum of the energy E_{copy} needed for transferring the data and the energy for executing the loop in Alg. 2. For each iteration, the loop needs an energy of $E_{\text{cpu}} + E_{\text{dram}}$ for processing workload on the CPU and E_{gpu} for the processing on the GPU. The constant energy needed by the code sections which do not depend on the workload distribution is neglected. Thus, the energy consumption is as follows:

$$E_{\text{cgm}} = l \left(E_{\text{cpu}} + E_{\text{dram}} + E_{\text{gpu}} \right) + E_{\text{copy}} \quad . \tag{27}$$

Due to the capability of the processing units to throttle down when they are in idle mode, there exist two different power values: P_{ex}, which is the power drawn when the processing unit is executing workload, and P_{idle}, which is the power drawn in idle mode, i.e. when no execution is possible. Hence, the energy values E_{cpu}, E_{dram} and E_{gpu} can be formulated as follows:

$$E_{\text{cpu}} = P_{\text{ex,cpu}} \cdot t_{\text{ex,cpu}} + P_{\text{idle,cpu}} \cdot t_{\text{idle,cpu}} \tag{28}$$

$$E_{\text{dram}} = P_{\text{ex,dram}} \cdot t_{\text{ex,dram}} + P_{\text{idle,dram}} \cdot t_{\text{idle,dram}} \tag{29}$$

$$E_{\text{gpu}} = P_{\text{ex,gpu}} \cdot t_{\text{ex,gpu}} + P_{\text{idle,gpu}} \cdot t_{\text{idle,gpu}} \quad . \tag{30}$$

In the following, each of the first summands in equations (28) to (30) is expressed by its energy value, i.e. $E_{\text{ex,cpu}}$, $E_{\text{ex,dram}}$ and $E_{\text{ex,gpu}}$, respectively.

The proportionality between E and n_{el} shown in formulae (8), (12) and (16) yield the equations

$$E_{\text{ex,cpu}} = p \cdot n_{\text{el,cpu}} \cdot E_{\text{el,cpu}} \tag{31}$$
$$E_{\text{ex,dram}} = p \cdot n_{\text{el,cpu}} \cdot E_{\text{el,dram}} \tag{32}$$
$$E_{\text{ex,gpu}} = n_{\text{el,gpu}} \cdot E_{\text{el,gpu}} \tag{33}$$
$$E_{\text{copy}} = n_{\text{el,gpu}} \cdot E_{\text{el,copy}} \ . \tag{34}$$

For brevity, let $P^*_{\text{idle,cpu}}$ denote the sum of the CPU and the DRAM idle power, i.e.,

$$P^*_{\text{idle,cpu}} = P_{\text{idle,cpu}} + P_{\text{idle,dram}} \tag{35}$$

and let $E^*_{\text{el,cpu}}$ denote the sum of the computation energy and the memory energy needed for processing one element on the CPU, i.e.,

$$E^*_{\text{el,cpu}} = E_{\text{el,cpu}} + E_{\text{el,dram}} \ . \tag{36}$$

Equation (27) for the energy of execution with both, CPU and GPU, can be reformulated as follows:

$$\begin{aligned} E_{\text{cgm}} = l \left(n_{\text{el,cpu}} \cdot E^*_{\text{el,cpu}} + n_{\text{el,gpu}} \cdot E_{\text{el,gpu}} \right) \\ + n_{\text{el,gpu}} \cdot E_{\text{el,copy}} \\ + P^*_{\text{idle,cpu}} \cdot t_{\text{idle,cpu}} + P_{\text{idle,gpu}} \cdot t_{\text{idle,gpu}} \ . \end{aligned} \tag{37}$$

4.3. Minimising the energy consumption

For executing the entire workload on the CPU, an energy of

$$E_{\text{cgm}} = l n_{\text{el}} E^*_{\text{el,cpu}} \tag{38}$$

is needed. The idle energy of the GPU can be neglected, since the GPU could be switched off or even removed from the machine.

Let $E^*_{\text{el,gpu}}$ denote the sum of the computation energy for one element on the GPU and the part of the data transfer energy attributed to one call of AXMEBE:

$$E^*_{\text{el,gpu}} = E_{\text{el,gpu}} + l^{-1} E_{\text{el,copy}} \ . \tag{39}$$

For executing the entire workload on the GPU with the CPU being idle, the energy consumed is the sum of the GPU energy and the idle energy of the CPU for the time of the execution:

$$E_{\text{cgm}} = l \left(n_{\text{el}} E^*_{\text{el,gpu}} + n_{\text{el}} t_{\text{el,gpu}} P^*_{\text{idle,cpu}} \right) \ . \tag{40}$$

Hence, for the decision how to distribute the workload, the following three cases have to be considered:

1. If $E^*_{\text{el,cpu}} < E^*_{\text{el,gpu}}$,
   then process all elements on the CPU.

2. If $E^*_{\text{el,cpu}} > E^*_{\text{el,gpu}} + P^*_{\text{idle,cpu}} \cdot t_{\text{el,gpu}}$,
   then process all elements on the GPU.

3. Otherwise,
   distribute the workload between CPU and GPU.

Symbol	Value	Remarks
$E_{\text{el,cpu}}$	265 µJ	in the *Turbo Boost* P-state
$E_{\text{el,dram}}$	20.5 µJ	in the *Turbo Boost* P-state
$E_{\text{el,gpu}}$	235 µJ	
$E_{\text{el,copy}}$	814 µJ	
$P_{\text{idle,cpu}}$	83 W	
$P_{\text{idle,dram}}$	1.3 W	
$t_{\text{el,gpu}}$	1.69 µs	
l	32.4	mean for 21 iterations of *crankshaft*

Table 2: Measurement values for quantities needed for the energy-aware workload distribution

When the workload is distributed between CPU and GPU, the idle times in equation (37) have to be minimised, since during the idle times, energy is consumed but no useful output is produced. Minimising the idle time means to ensure that $t_{\text{cpu}} = t_{\text{gpu}}$, i.e. finding the r with a minimal execution time according to equation (26). For $E_{\text{el,cpu}}$, the values for any CPU clock frequency may be used. If the condition for case 1 is satisfied for no frequency, and the condition for case 2 is satisfied only for some frequencies, the strategy of case 3 should be used with the frequency having the lowest $E_{\text{el,cpu}}$.

The values obtained in the experiments in Sect. 3 for the machine *Sandybridge* are shown in Tab. 2. For the mean number of CGM iterations, the mean value of the first 21 iterations with the *crankshaft* test object has been used. For other objects, the value of l might be anticipated for an FEM iteration by using the mean of the previous FEM iterations.Inserting the values from Tab. 2 into the equations (36), (39) and (35) yields the following values:

- $E^*_{\text{el,cpu}} = 286$ µJ,
- $E^*_{\text{el,gpu}} = 260$ µJ,
- $P^*_{\text{idle,cpu}} = 84$ W.

The values show that condition for case 1 is not satisfied. With the right-hand side of the condition of case 2, i.e. $E^*_{\text{el,gpu}} + P^*_{\text{idle,cpu}} \cdot t_{\text{el,gpu}}$, being 401 µJ, also the second condition is not satisfied. Hence, the third option, namely distributing the workload between CPU and GPU, has to be used for an energy-efficient execution of the CGM on the machine *Sandybridge*.

4.4. Discussion

The model predicts the execution time and the energy consumption of the CGM for p CPU cores and one GPU. Its extension to multiple GPUs is straightforward: If q is the number of GPUs, the total number of elements n_{el} is composed as follows:

$$n_{\text{el}} = p \cdot n_{\text{el,cpu}} + q \cdot n_{\text{el,gpu}} \quad . \tag{41}$$

The other equations have to be adapted accordingly. For example, the rate r of elements processed on the q GPUs is defined by the following equation:

$$r = \left(1 + \frac{t_{\text{el,gpu}}}{t_{\text{el,cpu}}} \cdot \frac{p}{q}\right)^{-1} \quad . \tag{42}$$

Other works that set up models for predicting the execution time of the CGM are, e.g. [39] and [40]: In [40], a sparse CGM is implemented on the GPU. An analytical model is presented which is used for optimising two implementation parameters: the number of threads and the size of the CUDA blocks. Parameters of the model are the warp size and the number of streaming processors of the GPU, which are machine-specific, as well as the length of each matrix row, which is data-specific. In [39] a model for executing a parallel CGM on multiple GPUs is set up. The model considers the dimension of the problem and the total number of stored elements in the matrix. The machine-specific parameters of the model are determined by multiple execution and a following fit.

The models of both works cited primarily rely on predicting the number of memory operations for predicting the execution time. They are more complex than the model set up in the present work, which is required due to the large differences in the matrices that may occur. In contrast, as all matrices occurring with the CGM investigated in the present work originate from an FEM, they are very similar. Remaining differences between matrices do not have to be modelled explicitly. Instead, differences still occurring can be covered by repeated measuring of the execution times with the online autotuning scheme described in the following section. As also Suda et al. state in [29], online autotuning requires incomplex models as only a limited number of parameters can be measured online without a negative impact on the performance of the application. Using online autotuning can also compensate possible deficiencies of the model.

5. Dynamic workload distribution

For minimising the energy consumption, the method for minimising the execution time by distributing the workload which has been presented in [10] is applied. The method redistributes the workload according to equation (26) in each refinement step of the FEM.

The dynamic distribution scheme works as follows: Each time, the routine AXMEBE is executed, its execution time on the CPU and on the GPU is measured. The measurement starts before and ends after the sections for t_{cpu} and t_{gpu} as marked by the vertical lines in Alg. 2. For multiple executions of AXMEBE with the same number of elements, i.e. during one execution of PPCGM, the means of the measured values for t_{cpu} and for t_{gpu} are calculated. These means are divided by the numbers of elements processed on the processing units, $n_{\text{el,cpu}}$ and $n_{\text{el,gpu}}$, to get the per-element times $t_{\text{el,cpu}}$ and $t_{\text{el,gpu}}$. With the values for $t_{\text{el,cpu}}$ and $t_{\text{el,gpu}}$ obtained in the previous refinement step, the new value r for the distribution of the workload between CPU and GPU is calculated according to equation (26). The first refinement step starts with $r = 0.5$.

Each refinement step adds more elements to the mesh. This increases the sizes of the data structures, i.e. the stiffness matrix, the load vector and the solution vector. Increasing data sizes may result in a different execution time behaviour, e.g. due to differences in the cache usage. The dynamic redistribution of the workload in each refinement step helps to retain the execution-time optimal distribution of the workload during the whole execution of the FEM. As shown in [10], no significant overhead is introduced into the application by the repeated measuring of execution times.

For processing elements on the GPU, the stiffness matrices for these elements are needed on the GPU as well as the full load vector and the current approximation of the solution vector. The respective data is copied to the GPU memory before the execution as shown in Alg. 2. Afterwards, the full result vector is copied back to the main memory.

This section investigates the behaviour of the dynamic workload distribution scheme on the machine *Sandybridge*, in addition to the machine *Westmere*, which has been investigated in [10]. The rate r of GPU elements is determined and it is investigated whether execution time and energy can be saved by exploiting the scheme.

5.1. Rate of elements processed on the GPU

In order to achieve a balanced distribution of workload, the condition of Eq. (24) has to be fulfilled: The execution times of the CPU and the GPU for AXMEBE should be roughly the same. Both times have been measured for the *drill hole* test object with the specific GPU rate r determined for each FEM iteration. Figure 11 shows that indeed the total times for processing the $n_{\text{el,gpu}}$ elements assigned to the CPU and the $n_{\text{el,gpu}}$ elements assigned to the GPU are roughly the same.

Figure 12 shows how the rate r of elements processed on the GPU for the two test objects changes during the mesh refinement steps. The chart shows that for the *Westmere* machine, the GPU rate r is roughly $\frac{1}{2}$, whereas it is roughly $\frac{1}{3}$ for the *Sandybridge* machine. Despite the fact that the results in Sect. 3.3 indicated that the execution times per element are roughly the same for CPU and GPU execution on *Sandybridge*, r is smaller than $\frac{1}{2}$ as for

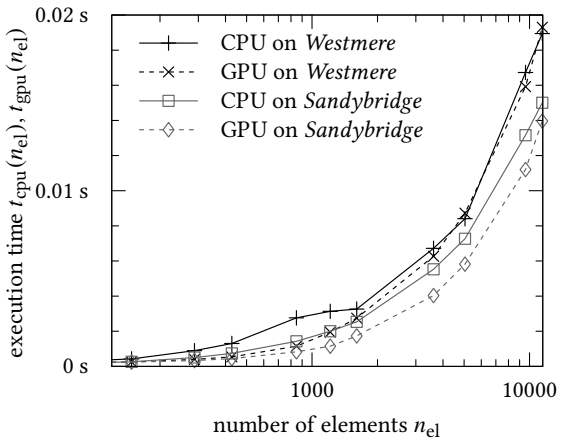

Figure 11: Comparison of the execution times of the CPU and the GPU for the *drill hole* test case

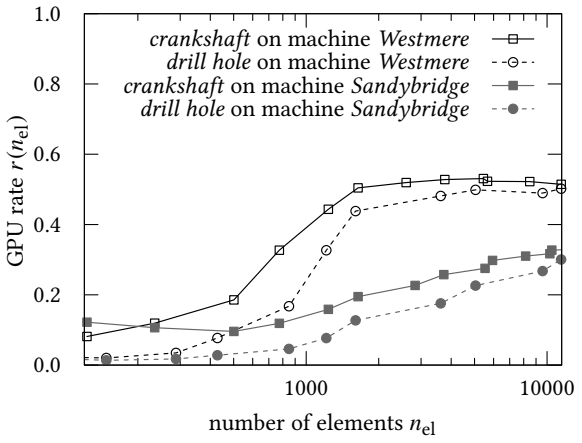

Figure 12: Rate r of elements processed on the GPU for two different hardware configurations and two different test cases

the workload distribution, the data transfer times have to be considered additionally. This fact leads to a decrease of the GPU rate.

The GPU rate is adapted during the execution of the FEM with increasing numbers of elements. On *Westmere*, the CPU threads need significantly more time for 400 to 1500 elements, see Fig. 11. This results in the fast increase in the GPU rate as clearly recognisable in Fig. 12. On *Sandybridge*, the CPU threads need slightly longer for nearly all numbers of elements resulting in a permanent increase of the GPU rate in Fig. 12.

5.2. Performance benefit

Figure 13 shows the summed execution time of the distribution-dependent sections of the routine PPCGM over all CGM iterations for the *crankshaft* test object. The measurement has been conducted on both machines for the three variants: CPU-only execution, GPU-only execution and CPU/GPU collaboration. The collaboration variant achieves a speed-up of roughly 25 % compared to the better of the CPU-only and the GPU-only variants for both machines. Whereas the GPU-only variant is better than the CPU-only variant on the *Westmere* machine, there is the opposite situation on the *Sandybridge* machine. An explanation could be found when considering the performance of the processing units: The peak performance of the CPUs in *Sandybridge* is twice the peak

performance of the CPUs in *Westmere*. Similarly, the peak performance of the GPU in *Sandybridge* is three times better. However, the memory speed of its GPU is worse which results in a lower data transfer bandwidth. As a consequence, the impact of the data transfer times is disproportionately large. Hence, on the *Sandybridge* machine, the CPU outperforms the GPU.

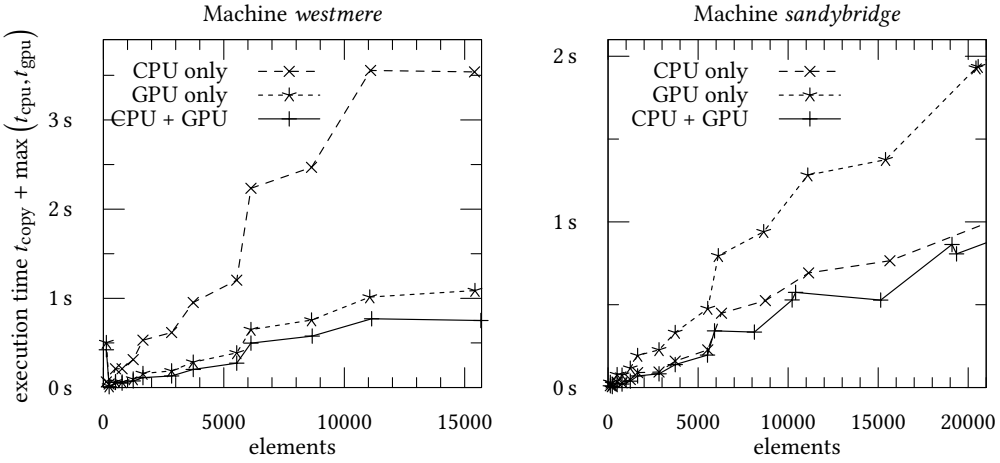

Figure 13: Execution times of CPU-only, GPU-only and CPU/GPU collaboration execution for the *crankshaft* test object

5.3. Discussion

In Sect. 5.1, it has been shown that for the *crankshaft* test object on the machine *Sandybridge* the rate r of elements processed on the GPU is roughly $\frac{1}{3}$. One iteration for this object with 10,000 elements needs an energy of 1.91 J on the CPU and an energy of 0.87 J for the computation and the data transfer on the GPU. In total, this yields an energy of 2.78 J. If only one CPU was used and the other was throttled down as suggested in Sect. 3.8, roughly one half of the elements would be processed on each, the GPU and one CPU, in order to maintain a good balance. This yields an energy consumption of 1.43 J for the CPU and 1.30 J for the GPU. With the idle power of the second CPU being 18 W, the idle energy is 0.15 J, which results in a total energy consumption of 2.88 J. Hence, for an execution on the machine the machine *Sandybridge*, it is advisable to use both CPUs for CPU/GPU collaboration.

6. Related work

This section presents related work. At first, some efficient implementations of the CGM on GPUs are presented. Then, a short overview on energy modelling in scientific computing is given. Finally, different works deploying a CPU/GPU collaboration are discussed.

CGM on GPUs

An energy-efficient CGM for GPUs is implemented in [41]. The energy efficiency of the implementation is improved by reformulating the CGM so that all computation is performed on the GPU. The prior implementation consisted of several GPU kernels with intermediate sections executed on the CPU, so that numerous synchronisations had to be performed.

Implementations of the CGM on multi-GPU environments without considering the energy are, e.g., [42] and [43]. In [42], the workload is distributed uniformly on the GPUs. In [43] a solver is chosen among a set of solvers considering the input data. By measuring the execution times of each solver for 3 iterations, the solver

yielding the shortest execution time can be identified and used for the rest of the execution. All approaches have in common that they use GPUs only—and no CPUs—for the computation, in contrast to the approach presented in this article.

Execution time and energy modelling

Execution time modelling has been a field of research for a long time. Good overviews on this subject are, for example, given in [44] and [45]. A vast number of modelling methodologies has been developed. Even frameworks for an automated modelling of the execution time of parallel algorithms exist, such as PACE [46] and Prophesy [47].

In contrast to execution time modelling, articles reporting on energy modelling are rare: [21] and [48] present energy models for generalised workload that take the number of cores of the target machine and the frequency of these cores into account. The model in [21] is verified with the *extrapolation method* for solving initial-value problems of ordinary differential equations. The model in [48] is verified with the *embarrassingly parallel* benchmark set. [49] sets up execution time and power models for the high-performance LINPACK automatically using regression analysis.

Not a software model, but a hardware model for a BlueGene/L-like machine which is tested against a sparse CGM is set up in [50]. The hardware model predicts execution time and power considering different characteristics of the machine including core frequency, voltage, cache sizes etc. Using the model, a linear relationship between frequency and execution time as well as a super-linear relationship between frequency and energy consumption is found thus supporting the findings of the present article.

In [51], the performance of the adaptive FEM including the CGM is investigated for different SMP machines. This includes the investigation of two different data distribution strategies for the CGM. However, the development of a precise performance model was not an objective of the article [51].

CPU/GPU collaboration

There exist various approaches for CPU/GPU collaboration: Harmony [52] is a programming and execution model which allows the coding of programs for CPU/GPU systems and their execution. The Harmony runtime system includes an automated workload distribution on the CPU and the GPU. However, most applications do not benefit significantly from the collaboration. The results have also shown that a dynamic work distribution is essential as the execution time is highly machine-dependent, which supports the findings of this article.

MapCG [53] is a MapReduce framework which allows jobs to be executed either on CPUs or on GPUs. For different experiments carried out, the speedup of a CPU/GPU execution compared to a GPU-only execution is always below 1.1, in many cases even below 1. This is attributed to the need to serialise/deserialise data for copying the intermediate data.

In [54], a Cholesky factorisation which is formulated as a directed acyclic graph of tasks is investigated. Some of these tasks can only be executed on the CPU or on the GPU, some on both. The distribution of tasks to CPU or GPU is fixed, i.e. cannot be adapted to their execution speed. [55] investigates a tile-based Cholesky and QR factorisation. The execution times of the kernels on the processing units depending on the tile sizes are measured during the installation process and stored in a library. Using this information, an optimal balance between CPU and GPU computation is found.

In [56], the DAGuE framework [57], which defines an algorithm as an assembly of tasks in a directed acyclic graph, is extended by GPU computing capabilities. Tasks are implemented by codelets. There can be multiple codelets for each task, each supporting a different hardware platform. The CPU/GPU collaboration is enabled by the capability of executing a codelet in different versions at the same time. Compared to GPU-only execution using the MAGMA library, this extended DAGuE framework achieves a speedup of 1.2.

[58] distributes the `dgemm` matrix multiplication to multiple CPU cores and one GPU using an execution time prediction. For calculating this prediction, a formula is developed which takes into account both computation and data transfer time. For the actual calculation of the data distribution between CPU and GPU, however, the data transfer time is, in contrast to the present article, omitted as it is dominated by the computation time. The

parameters of this formula are estimated on the basis of theoretical peak values and are not measured with the real hardware as in this article. Compared to GPU-only execution, the performance is improved by 35 % when adding a quad-core CPU.

A CPU/GPU collaboration scheme for an tomographic reconstruction application is presented in [59]. It uses a task pool for distributing the workload and achieves a speed-up of 1.4 . . . 1.7 compared to CPU-only or GPU-only execution. Scheduling methods for jobs which can be executed on both, CPUs and GPUs, are proposed in [60].

Most GPU/GPU collaboration schemes dismissed in the related work achieved a speedup of 1.15 to 1.35 which is in the same order of magnitude as the speedup achieved for the CGM in this article, in which up to 1.25 could be achieved. Except DAGuE and the task pool approach they do not support distributing the workload between CPU and GPU dynamically. However, not all algorithms, and especially not existing codes, can be easily transformed into a directed acyclic graph of tasks, as needed by DAGuE. Using a task pool for workload distribution is not feasible for the present application due to its significant overhead for the large number of small tasks.

7. Conclusion

In this article, an execution time and energy model for the parallel CGM used in an adaptive FEM has been developed. This model makes it possible to find the most energy-efficient distribution of the workload to the processing units, CPU and GPU, of a given machine. Though being not very complex, the model delivers a precise prediction of the execution time on CPU and GPU. This is because the machine- and data-dependent parameters of the model are updated periodically by measuring them online, i.e. during the regular execution of the CGM.

For the machine *Sandybridge* used in this work, processing one element of the *crankshaft* test object on the CPU needs an energy of 286 µJ including the energy consumed by the RAM. Processing one element on the GPU needs on average 260 µJ including the necessary transfer of the data to the GPU memory. Though the processing on the GPU seems to be more energy efficient than on the CPU, a GPU-only execution is not the most energy-efficient solution: For each element processed on the GPU, the CPU needs an energy of 142 µJ while being idle. Hence, the CPU/GPU collaboration using both kinds of processing units simultaneously is the most energy-efficient way of executing the parallel CGM in this case.

For the CPU/GPU collaboration, an online autotuning scheme has been exploited which minimises the energy consumption by minimising the execution time. The scheme measures execution times during each refinement step of the FEM. The workload distribution with a minimal execution time is determined by the prediction of the execution time model for the subsequent refinement step. The prediction always uses the latest measurement values. By having this feed-back, the distribution is adapted dynamically to influences which change during the execution of the application, for example with increasing data sizes. The scheme also ensures an automatic adaption to the underlying hardware.

The approach of dynamically distributing workload between processing units based on a model and online measurement presented in this article can be transferred to other problems or other platforms, e.g. different kinds of accelerators. It is especially suitable for applications whose runtime behaviour changes during the execution with a big variance in the properties of the data. Also for executing scientific applications in the cloud, an online autotuning approach is worth being considered: With cloud computing, one normally does not know the hardware platform on which the application is executed beforehand. Measuring the parameters of the model during the execution on the real machine with the real data allows to adapt the data distribution or other parameters of the implementation in order to make the execution faster or more energy-efficient. Furthermore, the repeated measuring helps to compensate possible deficiencies of the model. Online autotuning requires models with only a few parameters as not too many measurements should be made during the execution of the application for avoiding a negative impact on its execution time.

The experiments have shown that improvements in the measurement capabilities are needed for a more exact energy analysis. A higher temporal resolution of the energy MSRs in the CPU and the power meter of the GPU would be desirable. Furthermore, energy measurement would be simplified on the GPU if it offered an

interface for retrieving the energy, compared to the power interface currently existing. Such an interface could be implemented by integrating the power in hardware or in the GPU driver.

For supporting application developers, a tool automating the process and the experiments described in this article would be helpful. Such a tool could assist in finding laws for the energy consumption of the application considered or in identifying code sections offering potential for energy-efficiency improvements. A software tool could, for example, profile the energy consumption of an application as an execution-time profiler creates a profile for the execution time nowadays. With such supporting tools, the developer can constrain her- or himself to the "creative" work for which machines are unsuitable.

Acknowledgement This work is supported by the Deutsche Forschungsgemeinschaft under grants number RU 591/9-2 and RU 591/10-2 and within the Federal Cluster of Excellence EXC 1075 *MERGE Technologies for Multifunctional Lightweight Structures*. The source code of the FEM has been provided by the group Numerical Analysis of Arnd Meyer at the Department of Mathematics of the Chemnitz University of Technology, which the authors gratefully acknowledge. The Tesla C2075 GPU has been donated by Nvidia.

References

[1] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, V. Volkov, Parallel computing experiences with CUDA, Micro, IEEE 28 (4) (2008) 13–27.

[2] J. Nickolls, W. J. Dally, The GPU computing era, Micro, IEEE 30 (2) (2010) 56–69.

[3] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, S. Kato, Power and performance analysis of GPU-accelerated systems, in: Workshop on Power Aware Computing and Systems (HotPower'12), 2012.

[4] S. Huang, S. Xiao, W. Feng, On the energy efficiency of graphics processing units for scientific computing, in: IEEE Int. Symp. on Parallel Distributed Processing (IPDPS 2009), 2009, p. 1–8.

[5] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, M. Sarrafzadeh, Energy-aware high performance computing with graphic processing units, in: Workshop on Power Aware Computing and Systems (HotPower'08), 2008.

[6] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, K. Yelick, The international exascale software project roadmap, International Journal of High Performance Computing Applications 25 (1) (2011) 3–60. arXiv:http://hpc.sagepub.com/content/25/1/3.full.pdf+html.

[7] S. Marinkovic, hQ: From master/slave to masterpiece, [2013-11-11] (Oct. 2013).
URL `http://community.amd.com/community/amd-blogs/amd-business/blog/2013/10/21/hq-from-masterslave-to-masterpiece`

[8] S. Beuchler, A. Meyer, M. Pester, SPC-Pm3AdH v1.0 – Programmer's manual, Preprint SFB393 01-08, TU Chemnitz (2001, revised 2003).

[9] M. Balg, J. Lang, A. Meyer, G. Rünger, Array-based reduction operations for a parallel adaptive FEM, in: R. Keller, D. Kramer, J.-P. Weiß (Eds.), Facing the Multicore Challenge III, Vol. 7686 of LNCS, Springer, 2013.

[10] J. Lang, G. Rünger, Dynamic distribution of workload between CPU and GPU for a parallel conjugate gradient method in an adaptive FEM, Procedia Computer Science 18 (2013) 299–308, proceedings of the International Conference on Computational Science, ICCS 2013.

[11] J. Lang, G. Rünger, Measuring and modelling energy consumption for a CPU/GPU conjugate gradient method in an adaptive FEM, in: High-Level Programming for Heterogeneous and Hierarchical Parallel Systems 2014 Workshop, 9th HiPEAC conference, 2014.

[12] Intel microprocessor export compliance metrics: Intel Xeon processor 5600 series, [2013-11-11] (Sep. 2011).
URL http://download.intel.com/support/processors/xeon/sb/xeon_5600.pdf

[13] Intel microprocessor export compliance metrics: Intel Xeon e5-2600 product family, [2013-11-11] (Sep. 2011).
URL http://download.intel.com/support/processors/xeon/sb/xeon_E5-2600.pdf

[14] Nvidia Tesla C2075 companion processor, Data sheet, Nvidia (Sept. 2011).
URL http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf

[15] Intel, Intel Xeon processor E5-2650, [2013-11-11].
URL http://ark.intel.com/products/47922

[16] Intel, Intel Xeon processor x5650, [2013-09-06].
URL http://ark.intel.com/products/64590

[17] Nvidia, Geforce gtx 570 | specifications, [2013-11-11].
URL http:
//www.geforce.com/hardware/desktop-gpus/geforce-gtx-570/specifications

[18] S. Browne, J. Dongarra, N. Garner, G. Ho, P. Mucci, A portable programming interface for performance evaluation on modern processors, International Journal of High Performance Computing Applications 14 (3) (2000) 189–204.

[19] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, C. Le, RAPL: memory power estimation and capping, in: Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design, ISLPED '10, ACM, New York, NY, USA, 2010, pp. 189–194.

[20] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer's Manual (May 2012).

[21] T. Rauber, G. Rünger, Modeling and analyzing the energy consumption of fork-join-based task parallel programs, Concurrency and Computation: Practice and Experiencedoi:10.1002/cpe.3219.

[22] V. Gupta, P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan, G. Srinivasa, The forgotten 'uncore': On the energy-efficiency of heterogeneous cores, in: Proceedings of USENIX ATC 2012 Short Paper, 2012.

[23] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, E. Weissmann, Power-management architecture of the intel microarchitecture code-named sandy bridge, IEEE Micro 32 (2) (2012) 20–27. doi:10.1109/MM.2012.12.

[24] V. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, S. Moore, Measuring energy and power with PAPI, in: Int. Workshop on Power-Aware Systems and Architectures (PASA 2012), Pittsburgh, PA, 2012.

[25] D. Hackenberg, T. Ilsche, R. Schöne, D. Molka, M. Schmidt, W. E. Nagel, Power measurement techniques on standard compute nodes: A quantitative comparison, 2013 IEEE Int. Symposium on Performance Analysis of Systems and Software (ISPASS) (2013) 194–204.

[26] M. Kerrisk, The Linux Programming Interface, 1st Edition, No Starch Press, San Francisco, CA, 2010.
URL http://www.man7.org/tlpi

[27] Nvidia, NVML API Reference Manual, ver. 3.295.45 (2012).
URL http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/NVML/nvml.pdf

[28] J. Lang, G. Rünger, High-resolution power profiling of GPU functions using low-resolution measurement, in: F. Wolf, B. Mohr, D. an Ney (Eds.), Euro-Par 2013, no. 8097 in LNCS, Springer, Berlin, Heidelberg, 2013, pp. 801–812.

[29] R. Suda, L. Cheng, T. Katagiri, A mathematical method for online autotuning of power and energy consumption with corrected temperature effects, Procedia Computer Science 18 (2013) 1302–1311, proceedings of the International Conference on Computational Science (ICCS 2013).

[30] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, M. Stan, HotLeakage: A temperature-aware model of subthreshold and gate leakage for architects, Tech. rep. (2003).

[31] Nvidia, Tesla C2075 Computing Processor Board. Board Specification, bD-05880-001_v02 (2011).
URL http://www.nvidia.com/docs/IO/43395/BD-05880-001_v02.pdf

[32] M. Etinski, J. Corbalán, J. Labarta, M. Valero, Understanding the future of energy-performance trade-off via DVFS in HPC environments, Journal of Parallel and Distributed Computing 72 (4) (2012) 579–590.

[33] G. Balakrishnan, Understanding Intel Xeon 5500 Turbo Boost technology, How to use Turbo Boost technology to your advantage, IBM (Jun. 2009).

[34] Intel, Intel Turbo Boost technology—on-demand processor performance, [2013-10-25].
URL http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

[35] T. Mudge, Power: a first-class architectural design constraint, Computer 34 (4) (2001) 52–58. doi:10.1109/2.917539.

[36] J. Zhuo, C. Chakrabarti, Energy-efficient dynamic task scheduling algorithms for dvs systems, ACM Transaction on Embedded Computing 7 (2) (2008) 17:1–17:25.

[37] Samsung Electronics, 240pin Registered DIMM based on 2Gb D-die 1.35V: datasheet, rev. 1.2 Edition, [2013-11-25] (Aug. 2011).
URL http://www.samsung.com/global/business/semiconductor/file/2011/product/2011/9/6/476443ds_ddr3_2gb_d-die_based_1_35v_rdimm_rev12.pdf

[38] RAMpedia, Memory power consumption – DRAM IDD, [2013-11-25].
URL http://www.rampedia.com/index.php/ae2a

[39] M. Verschoor, A. C. Jalba, Analysis and performance estimation of the conjugate gradient method on multiple gpus, Parallel Comput. 38 (10-11) (2012) 552–575. doi:10.1016/j.parco.2012.07.002.
URL http://dx.doi.org/10.1016/j.parco.2012.07.002

[40] F. Vázquez, J. J. Fernández, E. M. Garzón, Automatic tuning of the sparse matrix vector product on gpus based on the ellr-t approach, Parallel Computing 38 (8) (2012) 408 – 420. doi:http://dx.doi.org/10.1016/j.parco.2011.08.003.
URL http://www.sciencedirect.com/science/article/pii/S0167819111001050

[41] H. Anzt, J. I. Aliaga, J. Perez, E. S. Quintana-Ortí, Reformulated conjugate gradient for the energy-aware solution of linear systems on GPUs, in: Proc. 2013 42nd International Conference on Parallel Processing, 2013, pp. 320–329. doi:10.1109/ICCP.2013.41.

[42] M. Ament, G. Knittel, D. Weiskopf, W. Strasser, A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-GPU platform, in: 18th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP), 2010, pp. 583–592.

[43] A. Cevahir, A. Nukada, S. Matsuoka, High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning, Computer Science - Research and Development 25 (2010) 83–91.

[44] L. Carrington, A. Snavely, N. Wolter, A performance prediction framework for scientific applications, Future Generation Computer Systems 22 (3) (2006) 336–346.

[45] H. A. Sanjay, S. Vadhiyar, Performance modeling of parallel applications for grid scheduling, Journal of Parallel and Distributed Computing 68 (8) (2008) 1135–1145.

[46] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, D. V. Wilcox, Pace–A toolset for the performance prediction of parallel and distributed systems, International Journal of High Performance Computing Applications 14 (3) (2000) 228–251.

[47] V. Taylor, X. Wu, R. Stevens, Prophesy: An infrastructure for performance analysis and modeling of parallel and grid applications, SIGMETRICS Performance Evaluation Review 30 (4) (2003) 13–18.
URL http://doi.acm.org/10.1145/773056.773060

[48] R. Ge, X. Feng, K. W. Cameron, Modeling and evaluating energy-performance efficiency of parallel processing on multicore based power aware systems, in: IEEE Int. Symposium on Parallel Distributed Processing, 2009. IPDPS 2009., 2009, pp. 1–8.

[49] B. Subramaniam, W. Feng, Statistical power and performance modeling for optimizing the energy efficiency of scientific computing, in: Proceedings of the 2010 IEEE/ACM Int. Conf. on Green Computing and Communications & Int. Conf. on Cyber, Physical and Social Computing, GREENCOM-CPSCOM '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 139–146.

[50] K. Malkowski, I. Lee, P. Raghavan, M. J. Irwin, Conjugate gradient sparse solvers: performance-power characteristics, in: Proceedings of the 20th Int. Conf. on Parallel and distributed processing, IPDPS'06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 297–297.

[51] J. Hippold, G. Rünger, Performance analysis for parallel adaptive FEM on SMP clusters, in: J. Dongarra, K. Madsen, J. Wasniewski (Eds.), Applied Parallel Computing – State of the Art in Scientific Computing, Vol. 3732 of LNCS, Springer, 2006, p. 730–739.

[52] G. F. Diamos, S. Yalamanchili, Harmony: an execution model and runtime for heterogeneous many core systems, in: 17th Int. symposium on High performance distributed computing, HPDC '08, ACM, New York, NY, USA, 2008, pp. 197–200.

[53] C.-T. Hong, D.-H. Chen, Y.-B. Chen, W.-G. Chen, W.-M. Zheng, H.-B. Lin, Providing source code level portability between CPU and GPU with MapCG, Journal of Computer Science and Technology 27 (2012) 42–56.

[54] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parallel Comput. 36 (5-6) (2010) 232–240.

[55] F. Song, S. Tomov, J. Dongarra, Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems, in: 26th ACM Int. Conf. on Supercomputing, ICS '12, ACM, New York, NY, USA, 2012, pp. 365–376.

[56] G. Bosilca, A. Bouteiller, T. Herault, P. Lemarinier, N. O. Saengpatsa, S. Tomov, J. J. Dongarra, Performance portability of a GPU enabled factorization with the DAGuE framework, in: IEEE Int. Conf. on Cluster Computing (CLUSTER), 2011, pp. 395–402.

[57] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, DAGuE: A generic distributed DAG engine for high performance computing, Parallel Computing 38 (1-2) (2012) 37–51.

[58] M. Fatica, Accelerating linpack with CUDA on heterogenous clusters, in: 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, ACM, New York, NY, USA, 2009, pp. 46–51.

[59] J. I. Agulleiro, F. Vázquez, E. M. Garzón, J. J. Fernández, Hybrid computing: CPU+GPU co-processing and its application to tomographic reconstruction, Ultramicroscopy 115 (0) (2012) 109 – 114.

[60] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, S. Chakradhar, Scheduling concurrent applications on a cluster of CPU-GPU nodes, in: 12th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2012, pp. 140–147.