MB3 D7.15– Final report on OpenMP and operating system adjustments and scheduling policies
Version 1.0

Document Information

Contract Number	671697
Project Website	www.montblanc-project.eu
Contractual Deadline	PM39
Dissemination Level	PU
Nature	Report
Authors	Daniel Ruiz (Arm), Roxana Ruşitoru (Arm)
Contributors	Daniel Ruiz (Arm), Roxana Ruşitoru (Arm)
Reviewers	Gundolf Hasse (UGRAZ), Jose Gracia (USTUTT-HLRS)
Keywords	Power-aware scheduling, heterogeneous compute, Arm big.LITTLE, Roofline Model

Notices: *This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N^o 671697.*

Change Log

Version	Description of Change
v0.1	Initial version of the deliverable
v0.2	Post-internal review version
v1.0	Final deliverable version

Contents

Executive Summary

In this deliverable, we revisit the heterogeneous device specifications (HDS) presented in deliverables D7.1 [Rus16] and D7.10 [Rus17a], in order to check inconsistencies or gaps in the HDS implementation.

We continue by proposing a new scheduling policy, adding it to those presented in deliverables D7.10 and D7.12 [Rus17b], based on the Roofline model and HDS. We cover the details of the application profile generation and the scheduling policy applied at application start-up based on which we decide which device to run on.

Results show that the Roofline Model scheduler achieves a high degree of accuracy, and could be even higher when applied to specific regions of code instead of application-wide.

We include a brief description of the Activity Monitoring Unit, an optional upcoming architectural feature that can assist with power management decisions.

1 HDS Revisited

Heterogeneous Device Specifications (HDS), which were introduced in D7.1 and expanded on in D7.10, are composed out of a set of Linux kernel and Device Tree Blob (DTB) modifications to provide further hardware information to the user. The information provided covers parameters from CPU frequencies to device-to-device (generally cores) latencies and bandwidth.

In order to improve the information offered, we revisited each of the metrics proposed on HDS v2.0 to look for possible improvements.

1.1 Heterogeneous Device Specifications v2.1

1.1.1 Original Fields on HDS v2.0

Table 1 shows the information present in HDS v2.0. This information is provided per each CPU core. Most of the information provided with HDS v2.0 is not available via any other way. In fact, this was the main target when designing HDS, to provide information about hardware capabilities that are usually hidden from the user.

Table 1: HDS v2.0 fields

Name	**Description**
`power`	List of instantaneous power consumptions for every available CPU frequency
`frequency`	List of available CPU frequencies
`max_dp_perf`	Maximum double-precision floating-point operations per second
`max_sp_perf`	Maximum single-precision floating-point operations per second
`pipeline_desc`	Specify whether a given CPU is in-order or out-of-order
`arch_capabilities`	List of architectural capabilities of the CPU (e.g., NEON, FPU, SVE)
`cache_latency`	Latency in cycles to access data currently present in L1
`mem_type`	Specifies the kind of memory used (e.g., DDR3, DDR4, HBM)
`mem_page_policy`	Specifies the current page policy
`mem_persistency`	Specifies the persistency capability of the memory
`mem_read_latency`	Cycles needed to read data located in memory
`mem_write_latency`	Cycles needed to write data in memory
`mem_topology`	Specifies the topology of memory (e.g., non-local, direct or local)
`mem_static_ppower`	Specifies the static power consumption of the memory
`mem_dynamic_ppower`	Specifies the maximum dynamic power consumption of the memory
`mem_channels`	Specifies the number of memory channels available
`mem_frequency`	Specifies the frequency at which memory is running
`mem_inter_core_latency`	Cycles needed to access data on another core within the same cluster
`mem_inter_cluster_latency`	Cycles needed to access data on another on a different cluster
`mem_inter_chip_latency`	Cycles needed to access data on another core on a differet SoC
`mem_inter_core_bandwidth`	Throughput when accessing data on another core within the same cluster
`mem_inter_cluster_bandiwdth`	Throughput when accessing data on another core on a different cluster
`mem_inter_chip_bandwidth`	Throughput when accessing data on another core on a different SoC

1.1.2 Modifications

Maximum single- and double-precision floating-point performance

In HDS v2.0, floating-point performance was a single FLOPS (i.e., floating-point operations per second) value. This was not reflecting reality, as floating-point performance depends on current CPU frequency. We had two options then:

1. Convert the entry to a list of values containing all the possible peak floating-point performances for every frequency.

2. Change it for a frequency-agnostic metric.

We decided to use the second option. Therefore, we created a new metric called *FLOPC*. FLOPC is defined as the floating-point operations per cycle. The actual value of this metric can be computed by using floating-point instruction latencies and throughputs for a specific architecture. The formula we use is the following:

$$FLOPC = FLOPI_X \times IPC_X \times P_X$$

where $FLOPC$ is the floating-point operations per cycle, $FLOPI_X$ is the floating-point operations performed by instruction X, IPC_X is the IPC of the instruction X and P_X is the number of pipelines that can execute the instruction X at the same time.

Obtaining the actual performance in FLOPS only requires a multiplication by the current frequency:

$$FLOPS = FLOPC * Freq$$

Where $FLOPS$ is the peak single-/double-precision floating-point operations per second, $FLOPC$ is the single-/double-precision floating-point operations per cycle of the CPU and $Freq$ is the CPU frequency in Hz (i.e., cycles per second).

2 HDS Scheduler

After the modifications performed to HDS, we decided to revisit again the HDS schedulers from deliverables D7.10 and D7.12. We added another scheduler policy which uses the Roofline Model [WWP09] to predict performance on the different cores, and therefore schedule the application on the more performant ones.

2.1 Roofline Model

The Roofline Model aims to predict the *roof performance* (i.e., the maximum performance you can obtain) of a given application running on a given platform. It uses a metric called *arithmetic intensity* (sometimes also called *operational intensity*) which is obtained by the following formula:

$$AI = \frac{FP_{ops}}{B}$$

Where AI is the arithmetic intensity, FP_{ops} is the number of floating-point operations and B is the number of Bytes that need to be accessed from memory by a given kernel. It is worth to notice that arithmetic intensity is actually the ratio between floating-point operations and memory accesses. Therefore, compute-bound algorithms will have a higher arithmetic intensity compared to the memory-bound ones.

Once you have a value of AI for your kernel, then you can predict the roof performance by applying the following formula:

$$P_{roof} = min(AI \times BW, R_{peak})$$

where P_{roof} is the *roof performance*, AI is the arithmetic intensity, BW is the memory bandwidth of the target platform (i.e., the platform where our code will be executed) and R_{peak} is the maximum single-/double-precision floating-point performance of the target platform.

Figure 1 presents the predicted roof performance for a set of operational intensities on a Juno r0 (2x Cortex-A57 and 4x Cortex-A53). This plot presents the predicted roof performance for different operational intensities. First thing to note is that this prediction is only regarding the peak performance, not real performance of real applications. For example, if a given application features an operational intensity value of 0.7, one should be able to know the maximum performance achievable by looking at the plot. Real executions could be much lower than predicted performance. The reason is that the Roofline Model only considers floating-point performance and memory bandwidth. If the studied application performance is bounded by any other characteristic, the current Roofline Model will not fit and therefore the predictions will not be accurate, needing to change how the operational intensity is computed (e.g., instead of floating-point operations use complex operations). Nevertheless, we decided to use floating-point operations as a metric for our study since all the applications chosen use it for their main kernels.

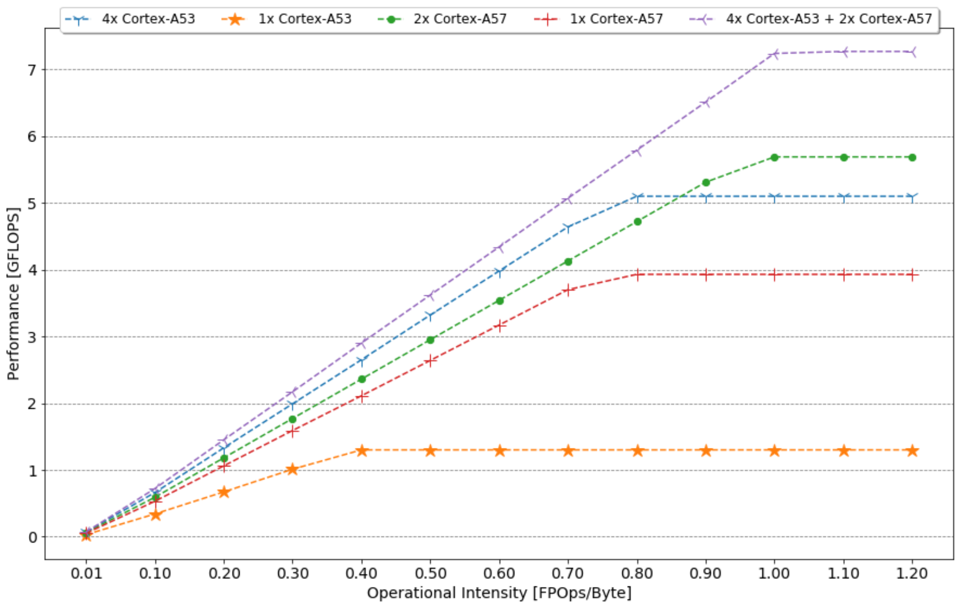

Figure 1: Predicted performance for different values of operational intensities on each of the clusters present at the Juno r0 SoC.

2.2 Hardware Platform

For our scheduling policy experiments, we use a Juno Arm Development Platform revision 0, which features the following hardware specification:

- 2x Cortex-A57 @ 1.10 GHz
- 4x Cortex-A53 @ 850 MHz
- 8GB DDR3
- 1x 1GbE network interface

For gathering application profiles, we used a Softiron Overdrive-3000 with the following specification:

- AMD Opteron A1100 SoC
  - 8x Cortex-A57 @ 2.00 GHz
- 16GB DDR3
- 1x 1GbE network interface

2.3 Software Stack

We use the following software stack for our experiments:

- HDS-enabled Linux kernel v3.15
- GNU Compiler Collection v7.3.0
  - GNU OpenMP
- Extrae v3.5.4
- Paraver v4.7.2

2.4 Application Profiles

We adjusted the application profiles to be used with the new Roofline Model scheduling policy. The new version includes a set of performance counters per parallel and serial regions. Each parallel or serial region of code is represented in the application profile separately, in a different line. Each parallel region is identified with a number in the interval $[0, N)$ where N is the number of parallel regions present in the code. The sequential region is identified with the value -1. For each of these regions we store the following hardware counters:

cycles: number of cycles.

floating-point instructions: number of executed SIMD and floating-point instructions.

memory instructions: number of load and store instructions.

load balance: a ratio that indicates if the parallel region is balanced (i.e., all the threads do the same amount of work) or not (i.e., some threads do more work than others). A value close to 1 means that the parallel region is balanced while a number close to 0 indicates that the parallel region is not well-balanced. Load balance is computed as:

$$LB_{PR} = \frac{T \times (\prod_{n=1}^{T} C_n)^{\frac{1}{n}}}{\sum_{n=1}^{T} C_n}$$

Where LB_{PR} is the load balance of parallel region PR, T is the number of threads used and C_n is the number of cycles of the thread n in the parallel region PR.

The application profile also contains the total number of cycles spent during the whole execution of the application. This information is used to compute the weight of a specific parallel region compared to the total execution of the application, therefore, the sum of the weights of all the regions must be equal to 1.

In comparison with the application profiles presented in deliverable D7.10, the new ones have been simplified in terms of number of performance counters, but gain more knowledge about the actual application structure.

2.5 Scheduler Policy

Our target is to design a scheduler that is able to decide whether it is better to run on the big cores (i.e., Cortex-A57), the LITTLE cores (i.e., Cortex-A53) or all of them depending on the arithmetic intensity of the different parts of the application. Actually, even though only Arm big.LITTLE architectures are currently supported, the objective is to support different kinds of devices.

The scheduler implements a modified Roofline Model in order to predict the performance of future execution. The main difference regarding the Roofline Model is that our scheduler does not know about operations but instructions. In this sense, arithmetic intensity cannot be exactly computed as it is defined by the model. Therefore, we use an approximation by defining that one floating-point instruction is equivalent to one floating-point operation. The same applies for memory instructions, which map directly to memory accesses.

The approximated arithmetic intensity then can be used in conjunction with the hardware information provided by HDS to predict the maximum performance per region. The formula to compute the final performance of a given cluster of cores is the following:

$$P_{roof} = \sum_{i=0}^{N}((min(FP_{par}, BW_{par} * AI_i) \times LB_i) + (min(FP_{ser}, BW_{ser} * AI_i) \times (1 - LB_i)))$$

where P_{roof} is the predicted roof performance, $FP_{par/ser}$ is the peak floating-point performance of one/all the cores of the big/LITTLE cluster, $BW_{par/ser}$ is the memory bandwidth of one/all the cores of the big/LITTLE cluster, AI_i is the arithmetic intensity of the region i and LB_i is the load balance of the region i. For the case of using all the cores, we empirically obtained the peak floating-point performance and the memory bandwidth by using micro-kernels. Apart from that, we use the performance of one LITTLE core for serial parts of the code since we observed that the Linux kernels tends to map serial parts of the code to the LITTLE cores when using both big and LITTLE cores.

Once the performance for the big and the LITTLE clusters and for all the cores at the same time has been computed, we compare the results and we decide to schedule on the one presenting the highest roof performance.

3 HDS Scheduler Evaluation

For all of our experiments, we use a set of mini-apps and benchmarks listed in Table 2, providing further details for each application.

Table 2: Applications used for testing the new scheduling policy

Application	**Description**
AMGMk [Law08]	Parallel algebraic multigrid solver for linear systems
CoMD [ExM13]	A classical molecular dynamics proxy application
graph500 [MWBA10]	Generation of, and Breadth first search through, an undirected graph
hpgmg-fv [ABS+14]	A proxy application for finite volume based geometric linear solvers
lulesh [KBK+13]	Livermore Unstructued Lagranian Explicit Shock Hydrodynamics
MCBenchmark [Law13]	A simple heuristic transport equation using a Monte Carlo technique
miniFE [HDC+09]	Proxy application for unstructured implicit finite element codes
HPCG [DHL15]	Preconditioned Conjugate Gradient method
HPCG-opt [RMC+18]	Custom version with HPCG's main kernel parallelized

We executed each of the applications on the Juno r0 system described in Section 2.2 using a different set of cores every time. The core configurations are:

- 4x Cortex-A53 (i.e., LITTLE cluster)
- 2x Cortex-A57 (i.e., big cluster)
- 4x Cortex-A53 + 2x Cortex-A57 (i.e., the whole SoC)

Figure 2 presents the relative performance from real executions for each of these configurations, using the 4xA53 performance as the baseline. In general, it is always recommended to execute on the big cluster, with a few exceptions. These exceptions are `hpcg-opt` (i.e., an optimized version of the HPCG benchmark), `CoMD` and `miniFE`, where it is better to use the whole SoC. The common characteristic between these applications is that all of them are memory bounded. In fact, the more memory bounded an application is, the more will benefit from executing on the all the cores available at the SoC since the achievable memory bandwidth is higher. The only exception is when different cores need to share large amounts of data. In this specific case, using both clusters at the same time incurs an overhead that is not hidden by the memory bandwidth increment.

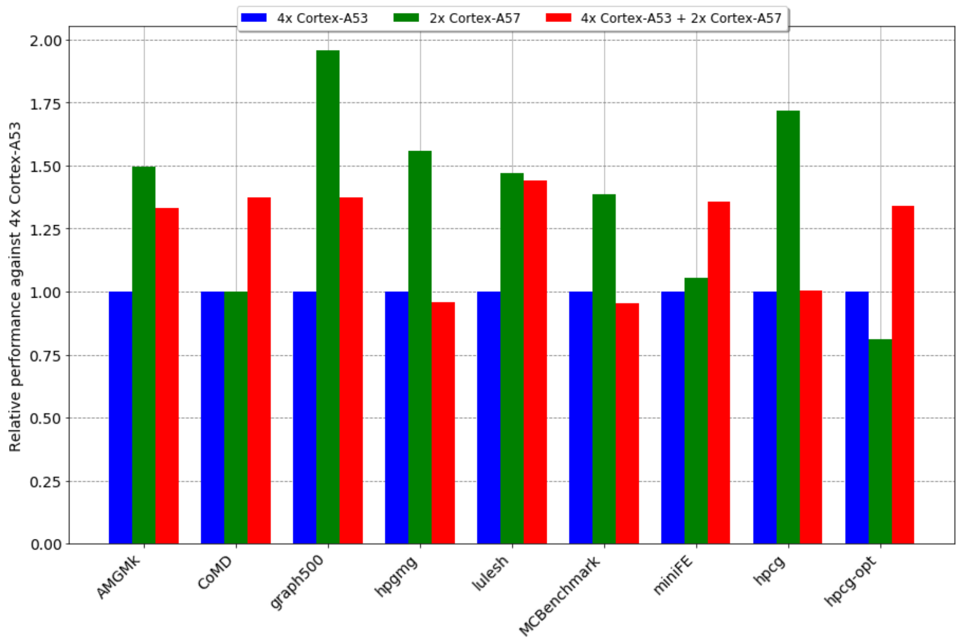

Figure 2: Relative performance of real executions of our set of applications when running on different cores of the same SoC

As for the rest of applications, they are more or less compute bound, therefore, better performance is expected (and measured) when using only the big cluster. Special mention to `hpcg` and `hpcg-opt`, where it can be seen that with the non-optimized version, it is better to use the big cluster, while when using the optimized code, one will get more performance by executing on all the cores or on the LITTLE cores compared to the big cores. The reference version of the HPCG benchmark (i.e., `hpcg`) is not parallelized during more than 70% of the execution time, therefore, it is better to use the big cores since they are the ones that provide better single-core performance both in terms of floating-point operations per second and memory bandwidth. The optimized version of HPCG (i.e., `hpcg-opt`) is actually parallelized across the whole execution. Since HPCG is heavily memory bounded, it benefits from using more cores so they can saturate the available memory channels.

We executed the same set of applications letting our scheduler choose which cores to use. Figure 3 shows the predicted relative performance of the applications compared to the predicted relative performance of the LITTLE cluster. The predicted performance is reported by the Roofline Model policy we implemented and considers the different weights and arithmetic intensities of the different parallel regions present in each of the codes.

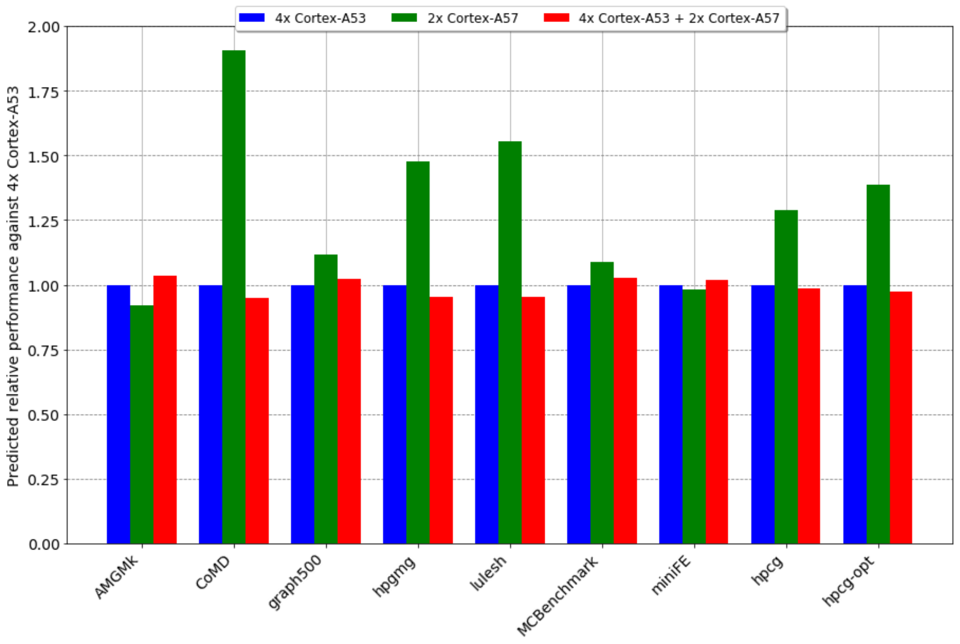

Figure 3: Relative predicted performance computed with our implementation of the Roofline Model for our set of applications

First thing to note is that the predictions for `AMGMk`, `CoMD` and `hpcg-opt` were not correct. We will study each case independently. As for the rest of applications, the predictions were accurate (i.e., the scheduler predicts higher performance when executing on the cores that actually provide more performance on real executions of the same application). This predicted performance is used by our Roofline Model policy to actually decide where the code will be executed (i.e., given an application, the scheduler will bind the OpenMP threads to the big or the LITTLE cluster, or to both, depending on which of those configurations presents the highest predicted performance).

As a final note, since our prediction is based on the Roofline Model, we should not expect to have the same differences in terms of predicted performance between different cores as in the real executions. The reason is that the Roofline Model actually predicts roof performance, not accurate performance, therefore, it can happen that the predicted performance is higher than the actual performance. This can be explain by the fact that the Roofline Model only considers two metrics (in our case, only floating-point operations and total number of bytes read/write from memory), so if the application is actually affected by any other metric, the final performance will be considerably lower than the predicted.

AMGMk

This mini-app executes three different kernels, `MATVEC`, `Relax` and `Axpy`. Each of these present different operational intensities, therefore, the three kernels could potentially fit better on a different cluster of cores. Our HDS scheduler, even though it considers parallel regions and their weights, makes an application-wide decision.

It is clear to see then that the kernel with the higher weight (i.e., more number of cycles compared to the rest) will have more weight on the decision. Figure 4 shows the execution time, from real executions, of each of the AMGMk kernels on the different core clusters. The kernel that spends more time to execute (i.e., more cycles) varies depending on the cores being used. Considering we generated our profiles by executions on a platform with only Arm Cortex-A57 cores, we should look at the corresponding performance. Therefore, we should choose to execute on the whole SoC, which provides the better performance for the `MATVEC` micro-kernel, even though it impacts negatively the performance of the rest of kernels.

This could be addressed by deciding which cores to use per parallel region, but this is not yet implemented in our scheduler.

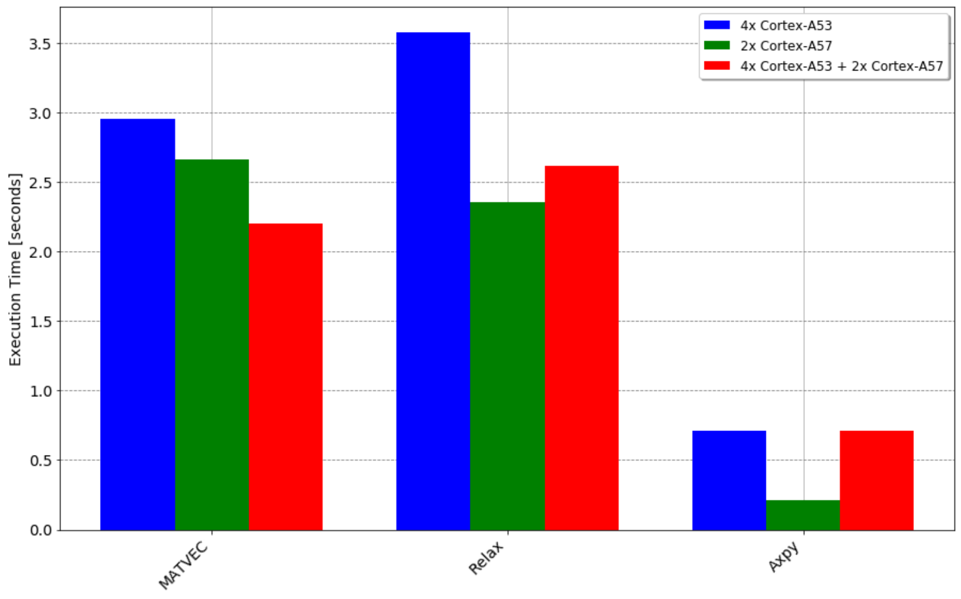

Figure 4: AMGMk performance by kernel

CoMD

The case of CoMD requires a deeper study. Whether the application is run on the big or LITTLE cores does not have a significant impact on performance. There is a slight benefit when running on all cores. This is not reflected in our model predictions. By looking at the profile, there are two main parallel regions that add for the 96% of the clock cycles. Both of them present an operational intensity of around 2, making them compute bound. Our experience tells that this kind of applications should perform better on the big cluster, and that is in fact reflected by our model. Therefore, there is something else we are missing. This is still under study.

HPCG-opt

The problem with this application is about the actual implementation of the benchmark. This benchmark consists of 6 phases:

- Problem setup and validation
- Compute reference sparse matrix-vector multiplication and symmetric Gauss-Seidel
- Compute reference conjugate gradient
- Setup optimized CG run and validation
- Optimized CG run
- Report results

The conjugate gradient is actually executed twice, once with the reference implementation and once with the optimized code. As the optimized code runs faster than the reference one, the reference implementation, which is composed mostly of serial code, has a higher weight. In fact, if we look at the `hpcg` performance, we will notice that it runs faster when only using the big cores. This is because the code is mainly serial, therefore if benefits from higher floating-point performance and memory bandwidth. Since performance is reported only by considering the optimized part, which actually benefits from using more cores that can offer a better memory bandwidth, the prediction is not the correct one.

4 Future Technology - Activity Monitoring Unit

The Activity Monitoring Unit is an optional architectural feature coming in the Armv8.4 architecture. It adds the option of dedicated core monitoring for power-performance management. This extension offers free-running read-only event counters accessible both by application software and firmware. The main goal of this unit is to offer, amongst others, a measurement of utilization and progress. These can then be used by power management software, such as improved power allocation.

The AMU comes with a small set of architected, fixed event counters:

- Running cycles at processor frequency
- Running cycles at constant frequency
- Instructions retired
- Memory stall cycles

In addition to those, there are auxiliary, implementation defined counters. Generally, these counters are 64-bit wrapping counters with overflow.

An early implementation of the upcoming AMU can be found in the Arm Cortex-A76 and Arm Cortex-A75 cores.

The AMU, unlike the Performance Monitoring Unit, offers a moderately-accurate system level overview of progress and utilization, whilst the PMU offers counters for detailed analysis of specific issues.

We envisage using such an architectural extension by dynamically detecting regions of code that stall, waiting for memory, and then migrating the affected threads to, for example, smaller cores (such as the LITTLE ones). We can also use the AMU to decide at runtime which device is better suited for a particular code, by starting say on big cores, and then migrating to the LITTLE ones, and then deciding on which is best. This method is not always beneficial, however, for codes with uniform behaviour, this could be applicable. One of the main motivations behind this is if one cannot obtain application profiles for a specific kernel (e.g., due to it not having been previously encountered).

5 Summary

In this deliverable, we investigated how the old HDS specification and scheduling policies, presented in deliverables D7.1, D7.10 and D7.12, behave and we then we modified them accordingly. For the former, we changed how floating-point performance was presented to the user, making the value frequency agnostic. Regarding the HDS scheduler, we implemented a new policy based on the Roofline Model which aims to predict the performance gains on the available devices of the target platform, binding the execution to the most fit device. This new policy achieves a high degree of success in its predictions, but it lacks deep knowledge of the application (e.g., applications were the figure of merit is derived from a part of the code what does not account for more than 50% of the total execution time).

There are still different aspects that need to be expanded in future developments. In this sense, we want to emphasize the lack of energy-awareness in the proposed HDS scheduler policy. We plan to include this feature to future iterations of the scheduler. Regarding the current HDS implementation, we are aware that it may lack useful information regarding devices different than CPUs or memory (e.g., if we want to include a GPU in our policy, we may need more information regarding this device, as latencies of data copies between the device and the CPU, the number of kernels that can be executed at the same time, etc.).

With the addition of the Activity Monitoring Unit in the Armv8.4 architecture, we can expand on the scope of what is trivially possible to achieve for power management, and we describe a few ways in which one can achieve improved power scheduling at runtime.

Acronyms and Abbreviations

- SoC: System-on-Chip
- HDS: Heterogeneous Device Specifications
- HPCG: High Performance Conjugate Gradient
- FLOPS: floating-point operations per second
- FLOPC: floating-point operations per cycle
- IPC: instructions per cycle

References

[ABS+14] Mark F. Adams, Jed Brown, John Shalf, Brian Van Straalen, Erich Strohmaier, and Samuel Williams. HPGMG 1.0: A benchmark for ranking high performance computing systems. Technical Report LBNL-6630E, Lawrence Livermore National Lab, 2014.

[DHL15] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. HPCG benchmark: a new metric for ranking high performance computing systems. Technical Report UT-EECS-15-736, University of Tennessee, Knoxville, 2015.

[ExM13] ExMatEx. CoMD: Classical molecular dynamics proxy application, 2013.

[HDC+09] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, HCarter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[KBK+13] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE Int. Parallel and Distributed Processing Symp.*, pages 1–14, 2013.

[Law08] Lawrence Livermore National Labs. ASC sequoia benchmark codes, 2008.

[Law13] Lawrence Livermore National Lab. Monte Carlo Benchmark, 2013.

[MWBA10] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 2010.

[RMC+18] Daniel Ruiz, Filippo Mantovani, Marc Casas, Jesus Labarta, and Filippo Spiga. The hpcg benchmark: analysis, shared memory preliminary improvements and evaluation on an arm-based platform. Technical Report, 2018.

[Rus16] Roxana Rusitoru. Mb3 d7.1 - initial report on heterogeneous device specifications, linux kernel and openmp capabilities. Technical report, Arm Ltd., 2016.

[Rus17a] Roxana Rusitoru. Mb3 d7.10 - intermediate report on alternative scheduling policies within openmp and the heterogeneous device specifications integration. Technical report, Arm Ltd., 2017.

[Rus17b] Roxana Rusitoru. Mb3 d7.12 - report of the investigation into alternative scheduling policies. Techincal report, Arm Ltd., 2017.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.