View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Hydraulic Engineering Repository

HENRY
Hydraulic Engineering Repository

Ein Service der Bundesanstalt für Wasserbau

Conference Paper, Published Version

Grasset, Judicaël; Longshaw, Stephen M.; Moulinec, Charles; Emerson, David R.; Audouin, Yoann; Tassi, Pablo

Porting TELEMAC-MASCARET to OPENPOWER and experimenting GPU offloading to accelerate the TOMAWAC module

Zur Verfügung gestellt in Kooperation mit/Provided in Cooperation with:
TELEMAC-MASCARET Core Group

Verfügbar unter/Available at: https://hdl.handle.net/20.500.11970/107159

Vorgeschlagene Zitierweise/Suggested citation:
Grasset, Judicaël; Longshaw, Stephen M.; Moulinec, Charles; Emerson, David R.; Audouin, Yoann; Tassi, Pablo (2019): Porting TELEMAC-MASCARET to OPENPOWER and experimenting GPU offloading to accelerate the TOMAWAC module. In: XXVIth TELEMAC-MASCARET User Conference, 15th to 17th October 2019, Toulouse. https://doi.org/10.5281/zenodo.3611548.

Standardnutzungsbedingungen/Terms of Use:

Die Dokumente in HENRY stehen unter der Creative Commons Lizenz CC BY 4.0, sofern keine abweichenden Nutzungsbedingungen getroffen wurden. Damit ist sowohl die kommerzielle Nutzung als auch das Teilen, die Weiterbearbeitung und Speicherung erlaubt. Das Verwenden und das Bearbeiten stehen unter der Bedingung der Namensnennung. Im Einzelfall kann eine restriktivere Lizenz gelten; dann gelten abweichend von den obigen Nutzungsbedingungen die in der dort genannten Lizenz gewährten Nutzungsrechte.

Documents in HENRY are made available under the Creative Commons License CC BY 4.0, if no other license is applicable. Under CC BY 4.0 commercial use and sharing, remixing, transforming, and building upon the material of the work is permitted. In some cases a different, more restrictive license may apply; if applicable the terms of the restrictive license will be binding.

Verwertungsrechte: Alle Rechte vorbehalten

Porting TELEMAC-MASCARET to OPENPOWER and experimenting GPU offloading to accelerate the TOMAWAC module

Judicaël GRASSET, Stephen M. LONGSHAW, Charles MOULINEC, David R. EMERSON
STFC Daresbury Laboratory
Warrington, United Kingdom
judicael.grasset@stfc.ac.uk

Yoann AUDOUIN, Pablo TASSI
EDF R&D
Chatou, France

***Abstract*—In this paper the state of porting TELEMAC-MASCARET on the OPENPOWER architecture with different compilers is shown. A port to GPUs with OpenMP and OpenACC of a computationally intensive subroutine of TOMAWAC is also explained and the performance benefits shown, a comparison with an x86-64 machine is also presented. Finally ongoing work is presented and discussed: the port of a complete and more challenging test-case, a triple coupling case from EDF.**

Keywords:OPENPOWER, POWER8, TOMAWAC, GPU, OpenACC, OpenMP

I. INTRODUCTION

Currently TELEMAC-MASCARET is parallelised with MPI, although attempts at hybrid parallelism have been tried in the past [1]. Improving the parallelisation of TELEMAC-MASCARET is useful for users who frequently perform simulations that take a long time to calculate. Current computer trends favour the increase of the number of cores in a single processor and, as shown by Fig. 1, this is being combined with the addition of accelerators such as GPUs, and memory interconnects designed to reduce the latency that is introduced by transferring data between different memory locations. It is therefore now important that TELEMAC-MASCARET is modified to take advantage of this new kind of architecure.

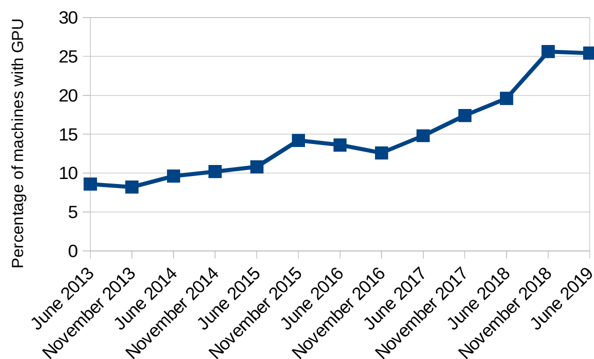

Figure 1. Evolution of the percentage of machines in the Top500 list that have GPUs in them.

There are two key options when choosing how best to run on GPUs, taking a low-level approach and programming directly with OpenCL or CUDA, or using pragma-based programming with OpenMP or OpenACC. The first option gives more control and usually more performance but it also means that a specific code has to be written and that two different versions of the same kernel have to be maintained. However, when using the pragma-based approach, changes to the code do not infer a re-write of the kernel, with the bulk of the changes needed being the addition of pragmas around the existing code. This approach reduces the burden on those that maintain the original codebase and means acceptance of changes is more likely. This work therefore concentrates on enabling GPU acceleration of portions of TELEMAC-MASCARET using a pragma-based approach.

This paper first presents a kernel of Tomawac ported to GPUs using OpenMP and OpenACC pragmas and tested on OPENPOWER and x86-64 architecture. The possibility of using GPUs acceleration on a more challenging test-case provided by EDF is then explored.

II. RELATED WORK

An attempt to use GPUs with TELEMAC-MASCARET has already been made [1], however the method presented is different from the one described in this article. In [1], the authors replaced the original matrix-vector product of TELEMAC-MASCARET with the one from the MAGMA library [2], which is then able to be offloaded to GPU.

The primary problem they encountered was that the MAGMA library was not using the same matrix format as TELEMAC-MASCARET. Doing the conversion before and after every matrix-vector product prevented any real-world performance improvement. This work shows how directly accelerating the existing code without modifying the data structure used by TELEMAC-MASCARET is a better approach.

III. MACHINES USED

A. Paragon

The OPENPOWER foundation [3] is a consortium of entities working to provide an architecture revolving around the POWER processors and accelerators. In this work the

architecture used consists of IBM POWER8 processors and NVIDIA GPUs. The processors are interfaced to the GPUs with NVLink instead of PCI-Express. NVLink is a high-bandwidth proprietary interface developed by NVIDIA [4], it is also used here to enable GPU to GPU interconnection.

This work has almost entirely been done on the UK Research and Innovation Science and Technology Facilities Council's (UKRI-STFC) Paragon POWER8 cluster, maintained and run by the Hartree Centre [5] at Daresbury Laboratory in the UK. Each node of the cluster consists of 2 POWER8 CPUs, each with 8 physical cores (up to 8 hardware threads per core) and 4 NVIDIA P100 GPUs with NVLink 1.0 interconnects. Each P100 has 16GB of memory and the 2 POWER8 CPUs share 1TB of memory.

B. *Wilkes-2*

The Wilkes-2 cluster [6] is hosted at Cambridge University, in the UK. Each node of the cluster consists of one Xeon E5-2650 v4 2.2GHz CPU with twelve physical cores, and four NVIDIA P100 GPUs. Each P100 has 16GB of memory and the node has 96GB of memory.

The Wilkes-2 cluster has the same GPUs as the Paragon cluster, so little difference in computation time is expected between them. However, the Paragon cluster has NVLink to transfer data between the CPUs and GPUs, while Wilkes-2 has standard PCI-Express which means that in the case of high-volume data-transfer between CPUs and GPUs the Paragon cluster has an advantage.

IV. PORTING TO THE OPENPOWER ARCHITECTURE

Before working on accelerating parts of TELEMAC-MASCARET using GPUs, it was first necessary to port and compile it using the OPENPOWER architecture. During testing, a significant bug with GCC for OPENPOWER was discovered, through this work this has been reported [7] and fixed in GCC 9.1. Similarly, a number of internal bugs within IBM's XL compiler have been found by this work, reported and fixed in the version 16.1.1.1. A single compilation issue when using the XL compiler remains in the current TELEMAC-MASCARET code-base but is easily rectified with a small patch. Finally, the PGI compiler tested is able to compile the lastest stable version of TELEMAC-MASCARET (v8p0r2) but fails when compiling the trunk, this has been reported on the PGI bug tracker [8].

Further problems with compilation using all three compilers were found when using OpenACC and OpenMP, these have all been reported to the respective public bug trackers for GCC [9] and PGI [10] and internally to IBM.

Version	≥ PGI 18.10	≥GCC 9.1	≥XL 16.1.1.1
v8p0r2	Compile	Compile	Does not compile[a]
trunk	Does not compile[a]	Compile	Does not compile[a]

a. Compilation possible following application of patch to TELEMAC-MASCARET

Table 1. Summary of the current state of TELEMAC-MASCARET on OPENPOWER with different compilers.

V. PORTING A KERNEL TO GPUS

A. *Kernel Choice*

In order to determine whether there would be any significant benefits to porting parts of TELEMAC-MASCARET to GPUs, a specific test-case was chosen. The case *fetch_limited/tom_test6.cas* of the wave propagation module TOMAWAC was a good candidate because it is computationaly intensive and most of the computations are localised in a single subroutine. Even though it is a complete test case with initialisation, finalisation and calls of numerous subroutines, 95% of the computational time is spent in the *qnlin3* subroutine. Any benchmarks of this kernel shown will give the execution of the whole test case and not only the time of the *qnlin3* subroutine. The execution time is measured using the internal timers of TELEMAC-MASCARET.

The *qnlin3* subroutine is approximately 400 hundred lines long and mainly consists of a four-level nested loops and updates to two three-dimensional arrays. Each array can have its cells updated multiple times during a single call of the subroutine.

As the original test-case mesh is very small, it was refined twice in order to increase the computation time. This was achieved with Stbtel and the *python* scripts from the TELEMAC-MASCARET suite. The final mesh was made of 75 664 elements and 32 127 points. Some parameters in the *.cas* steering file have also been changed: “NUMBER OF TIME STEP” was increased to 400 and “TIME STEP” to 225.

B. *ORIGINAL EXECUTION TIME*

Each core of the POWER8 CPU is able to work at different levels of Simultaneous Multi-Threading, (SMT1, SMT2, SMT4 and SMT8). This means that each core can execute more than one thread at the same time, e.g. two threads with SMT2. This functionality is comparable with the Hyperthreading technology of Intel processors. While Intel's Hyperthreading can only currently be used to run a maximum of two threads in parallel, a POWER8 core is able to run up to eight. Benchmarks have shown that TELEMAC-MASCARET does not benefit from the use of SMT8 (maybe because the memory bandwith is saturated, also SMT8 is not on par with SMT2 or SMT4 as it deactivates the CPU's instruction prefetcher [11]). As standard TELEMAC-MASCARET uses MPI parallelisation and is able to run on thousands of cores [12], early tests have showed that it is always beneficial to use SMT2 and in some cases SMT4 but never SMT8. This work therefore presents results using SMT1, SMT2 and SMT4.

Tables 2, 3 and 4 show that there is a significant difference in execution time for the same code, when compiled with different compilers but using the same basic optimisation parameters. The PGI compiler generates the fastest code, the IBM compiler produces code between 1.10 and 1.49 times slower than PGI and the GCC compiler generates code between 1.04 and 1.89 times slower than PGI. The biggest difference in execution time is noted when the

code is run on one or two nodes, while the smallest comes when the code is run on eight nodes.

Number of nodes	SMT1 execution time (s)	SMT2 execution time (s)	SMT4 execution time (s)
1	8442	6801	**6072**
2	4494	3376	**3172**
4	2240	1775	**1747**
8	1185	**980**	2489

Table 2. Execution time (s) comparison of the original code with different level of SMT when compiled with PGI.

Number of nodes	SMT1 execution time (s)	SMT2 execution time (s)	SMT4 execution time (s)
1	12 045	10 507	**8108**
2	6711	4274	**4186**
4	2697	**2254**	2276
8	1425	**1236**	2734

Table 3. Execution time (s) comparison of the original code with different level of SMT when compiled with IBM XL.

Number of nodes	SMT1 execution time (s)	SMT2 execution time (s)	SMT4 execution time (s)
1	15 973	9976	**7640**
2	6084	4338	**3899**
4	2865	2136	**2075**
8	1353	**1146**	2595

Table 4. Execution time (s) comparison of the original code with different level of SMT when compiled with GCC.

As shown in Tables 2,3 and 4, the performance difference between the compilers reduces as the number of MPI processes increases (and therefore the number of computations per MPI process decreases), therefore it can be hypothesised that the PGI compiler produces the fastest executable because of a better ability to vectorise the code. The vectorisation achieved by the PGI compiler appears efficient when there are a lot of computations per MPI process but when the processes have a small amount of work it does not make much difference.

C. OpenACC

OpenACC is an open standard set of directives to offload computations on GPUs. Between the three compilers used for this work, only PGI and GCC provide an OpenACC implementation.

The modifications introduced to use OpenACC for GPUs with the TOMAWAC module required only small changes to the code. The key change is that, in order to get good performance, the four loops have been moved closer to each other and have been collapsed (as seen in code sample 1). However this revealed a problem when compiling with PGI. Collapsing the four loops meant the compiler replaced the four loops with a new one which iterates from 1 to the multiplication of the four upper-bounds. In the original code the four max variables were 32 bit integers however, 32 bits is not enough to hold the multiplication of the four upper-bounds in the test-case used. The type has therefore been changed to a 64 bit integer. The problem has been mentioned in the PGI forum and they have proposed that the next version of the compiler will automatically use 64 bit integers when collapsing loops as an optimisation [10].

As the cells arrays can be updated several times during a single call to *qnlin3*, it was necessary to protect each update in order to make sure that no cells were updated simultaneously. To do so, each update is put in an atomic operation. In a pure CPU implementation this would be considered a bad approach as atomic instructions are typically slow but here GPU performance implications appear minimal. On CPU, instead of using atomic it would have been possible to use a reduction on the arrays, at the cost of added memory usage.

Finally, in OpenACC there is no directive to distribute a loop accross multiple GPUs. However, as TELEMAC-MASCARET is already parallelised with MPI we can easily take advantage of this. During the initialisation of TELEMAC-MASCARET each MPI rank is assigned to a GPU. When the process encounters a portion of code to offload, it sends it to the GPU it has been assigned to (assignement is maintained for the duration of the execution). E.g. when 4 MPI ranks are created, GPU 0 will be assigned to MPI rank 0 and GPU 1 to rank 1 and so on. When 8 MPI ranks are created, each GPU will have 2 MPI ranks assigned to it (since there are 4 GPUs on each node).

To validate that the modified code still gives the correct results, the result file is compared to the result file generated by the original code. In the benchmarks presented hereafter no differences between the two result files have been found.

Tables 3 and 4 show the results for the program compiled with the PGI and GCC compilers. It can be seen that the PGI implementation is the more efficient as it outperformed GCC by around three times. If these results are compared with the original execution time using MPI-only, then it shows that using GPU with OpenACC is highly beneficial. The program using GPU and compiled with the PGI compiler is between 4.4 and 5.5 times faster than the CPU MPI-only version and between 1.8 and 2.1 times faster for the program compiled with the GCC compiler.

<table>
<tr><td>

```
do i=1,maxi
!some operations
  do j=1, maxj
    !some operations
      do k=1, maxk
        !some operations
          do l=1, maxl
            arr(j,k,l) = arr(j,k,l) +1
```

</td><td>

```
!$acc parallel loop collapse(4)
do i=1,maxi
  do j=1.maxj
    do k=1,maxk
      do l=1,maxl
        !some operations
        !$acc atomic update
        arr(j,k,l) = arr(j,k,l) + x
```

</td></tr>
</table>

Code sample 1. Comparison of of the original and after transformation of a simplified part of *qnlin3*.

Number of nodes	Best original CPU execution time (s)	GPU (OpenACC) execution time (s)	Speedup (CPU / GPU)
1	6072	1367	4.4
2	3172	686	4.6
4	1747	342	5.1
8	980	179	5.5

Table 5. Comparison between CPU and CPU+GPU execution time with OpenACC when compiled with PGI.

Number of nodes	Best original CPU execution time (s)	GPU (OpenACC) execution time (s)	Speedup (CPU / GPU)
1	7640	4192	1.8
2	3899	2131	1.8
4	2075	1083	1.9
8	1146	554	2.1

Table 6. Comparison between CPU and CPU+GPU execution time with OpenACC when compiled with GCC.

D. OpenMP

Since version 4.0, OpenMP has offered its own GPU offloading capabilities similar to those provided by OpenACC, again these are pragma-based. Even though the pragmas are syntactically different from those in OpenACC, the ones used for offloading are functionally equivalent. The OpenMP offloaded version of *qnlin3* is therefore very similar to the OpenACC one (see code sample 2).

As the PGI compiler used only supports OpenMP pragmas for CPU, the IBM compiler has been used to evaluate OpenMP GPU offloading performance. GCC also implements OpenMP GPU offloading but for unknown reasons the program always crashes when entering the GPU code, it has therefore been impossible to test it so far.

Table 7 shows the results for the OpenMP offloading compared to the original MPI version, the two being compiled with the IBM XL compiler. It can be seen that there is still a notable acceleration when using the GPUs. On two nodes, the version running on GPUs is three times faster than the original MPI version and it is four times faster on eight nodes. However the speedup achieved is smaller than the one achieved with OpenACC. In fact, the OpenMP version is about two times slower than the OpenACC version compiled with PGI. This difference in performance could be attributed to having to use the IBM compiler rather than the PGI compiler used for the OpenACC tests as the IBM compiler typically produces slower code (as can be seen in Tables 2 and 3).

Performance on 4 MPI ranks has been impossible to measure. The problem is the same as described in section IV,C. The difference is that using a 64 bit integer as an index loop does not solve the problem, the IBM XL compiler seems to still generate GPU code which uses 32 bit integer. This problem has been reported to IBM.

```
!$omp target teams distribute parallel do collapse(4)
do i=1,maxi
  do j=1.maxj
    do k=1,maxk
      do l=1,maxl
      !some operations
      !$omp atomic update
      arr(j,k,l) = arr(j,k,l) + x
```

Code sample 2. OpenMP offloaded version of a simplified *qnlin3*.

E. Multiple MPI processes per GPU

Profiling using Nvidia's nvprof utility shows that the PGI compiled OpenACC implementation uses about 25% of the total occupancy of each GPU. Furthermore, the kernel does not run continously but only for about 60% of the total execution time of the whole program, so the GPU alternates between idle time (40%) and computing time (60%). In theory it should therefore be possible to run nearly 8 instances of the code on the GPU before hitting 100% usage for the test case shown. Also, as this case has a low memory consumption, there are no foreseeable problems running 4 instances of the code on one GPU from a memory consumption perspective. An added beneficial consequence of running four instances of the code on each of the GPUs means that every core of the Power8 CPU is also used when SMT1 is assumed.

A profiling of the version compiled with GCC has been done and was notably different from the PGI one. With PGI the kernels were taking 25% of the GPU computational capability and running for 60% of the time, with GCC the kernels use only 12.5% of the GPU capacity and are running for 83% of the time. This shows that PGI, at least in this case generates kernels which are able to extract more parallelism and use more of the computational power of the GPU.

Tables 8 and 9 show the results for using multiple MPI processes per GPU (up to one MPI process per core of the CPU). These results demonstrate that it is beneficial to run multiple instance of the code on the GPU compiled with the PGI compiler, the code benefits from an acceleration between 1.25 and 1.49. However when the code is compiled with GCC, the acceleration (between 1.03) is almost non-existant when running multiple instances of the code on the same GPU.

The same test has also been tried with the OpenMP version of the offloading. Table 10 shows the results. As with the PGI+OpenACC version, the IBM+OpenMP version can also benefit from offloading mulitple MPI processes on a GPU, the acceleration obtained is between 1.20 and 1.27.

Number of nodes	Best original CPU execution time (s)	GPU (OpenMP) execution time (s)	Speedup (CPU / GPU)
1	8108	Crash	–
2	4186	1401	3.0
4	2254	686	3.3
8	1236	336	3.7

Table 7. Comparison between CPU and CPU+GPU execution time with OpenMP when compiled with IBM XL.

Number of nodes	1 MPI per GPU execution time(s)	2 MPI per GPU execution time(s)	4 MPI per GPU execution time(s)
1	1367	1192	1090
2	686	612	532
4	342	303	253
8	179	146	120

Table 8. Comparison of execution time (s) when offloading multiple MPI processes on the same GPU with OpenACC when compiled with PGI.

Number of nodes	1 MPI per GPU execution time(s)	2 MPI per GPU execution time(s)	4 MPI per GPU execution time(s)
1	4213	4192	4086
2	2131	2115	2061
4	1083	1079	1051
8	554	553	539

Table 9. Comparison of execution time (s) when offloading multiple MPI processes on the same GPU with OpenACC when compiled with GCC.

Number of nodes	1 MPI per GPU execution time(s)	2 MPI per GPU execution time(s)	4 MPI per GPU execution time(s)
1	Crash	2533	2191
2	1401	1207	1098
4	686	603	542
8	336	302	280

Table 10. Comparison of execution time (s) when offloading multiple MPI processes on the same GPU with OpenMP when compiled with IBM XL.

F. Comparison with an x86-64 machine

This section provides results using a more typical x86-64 based cluster called Wilkes-2. It uses the same P100 GPUs as Paragon, but it does not have an NVlink interconnect between the CPUs and GPUs, this means that data transfers will be slower than on Paragon. As Table 11 shows, there is no significant differences in execution time between Wilkes-2 and Paragon when the code is offloaded to GPUs when using 1 to 4 nodes, this is most likely because data transfers in the presented case are small and infrequent. Currently it can be seen that when 8 nodes are used, execution time starts to rise again indicating a drop-off in scalability. However, given past experience and performance achieved using the Paragon Power8 system, this is unexpected behavior and may be attributed to a functional problem with the Wilkes-2 system. Further investigation of this problem will be undertaken.

VI. Porting a challenging test-case to GPUs

In the *fetch_limited/tom_test6.cas* test-case the performance bottleneck was the *qnlin3* subroutine, taking about 95% of the execution time. So the offloading to GPU was relatively simple with only one subroutine to offload in order to get good performance. Our ongoing work now focuses on porting more subroutines to GPU to enable accelerated calculation of more complex cases.

Number of nodes	Original, 12 MPI per node, execution time (s) and speedup	4 MPI, 4GPU per node, execution time (s) and speedup (cpu/gpu)	12 MPI, 4 GPU per node, execution time (s) and speedup (cpu/gpu)
1	17 339	1319 (13.1x)	1199 (14.5x)
2	8088	652 (12.4x)	650 (12.4x)
4	3751	328 (11.4x)	320 (11.7x)
8	1900	564 (3.4x)	477 (4x)

Table 11. Comparison of execution time (s) between original code and offloading to GPUs with OpenACC on Wilkes-2 when compiled with PGI.

EDF have provided a test case named *Somme_7days*. This is a triple coupling case using TELEMAC2D, SISYPHE and TOMAWAC. After profiling the code, it seems that most of the time is spent in Tomawac, but not in a single subroutine as with the presented test-case. In *Somme_7days* there is no clear bottleneck to note. The profiling has been done with the Linux perf profiler [13], using one core of one POWER8 node and the code was compiled with the PGI compiler.

In Fig 2. it can be seen that the main time-consuming subroutines are *schar41_per_4d*, *log*, *qnlin1* and *bief_interp*. It should be noted that *log* is mostly called in *qwind1*, thus their execution times could be merged. The main difference with the previous work done on the *tom_test6.cas* case is that this time it is not possible to offload one subroutine to GPU and get a significant acceleration, because no subroutine dominates. Another difference is that *qnlin3* was computationally expensive, but in this case no subroutine is equally as expensive, for instance a single call to *qnlin1* takes about 400 ms and a call to *bief_interp,* which is the most time consuming subroutine, takes about 20 ms. The subroutine *bief_interp* is the most expensive not because of the computational cost but because it is called very frequently during the execution of the program.

In order to get performance improvements with GPU offloading in this case, it will be necessary to offload multiple subroutines and since the subroutines are quick to execute and called thousands of times it is very important to minimise the data transfers between host machine and GPU. To achieve this it will be necessary to perform all transfers at the caller level and not within the offloaded subroutine. Doing so will introduce another complication, for instance it is easy to do the data transfers in *semimp* for all the offloaded subroutines that *semimp* will call. However if those subroutines are called outside of *semimp* then the data needed will not be available on the GPU, leading to a crash.

One solution would be to add a call to *acc_is_present* (or *omp_target_is_present*) at the beginning of each offloaded subroutine, if the data is present the code will be executed on the GPUs and if not on the CPU. But doing so does not take into account that some subroutines might be modified to execute more efficiently on GPUs (like a collapse of the loops, as seen on code sample 1) and that these modifications are usually guarded with a compile-time *ifdef* in order to not duplicate the code and keep the two versions in the same file.

Since only one part of the code guarded by the *ifdef* will be compiled it will not be possible to select it arbitrarily at runtime. An elegant solution to this problem is still being considered.

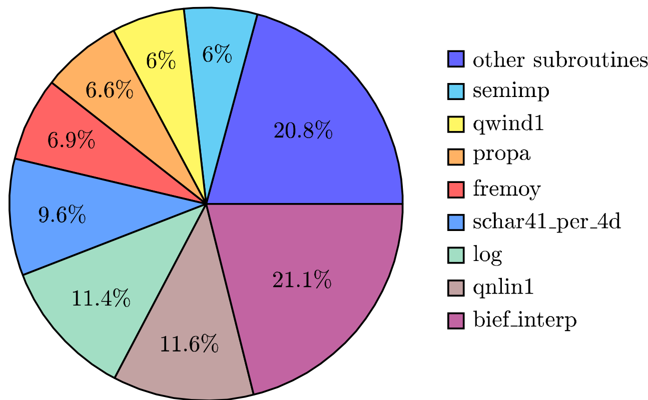

Figure 2. Profiling of the *Somme_7days* test-case (1 MPI, 1 POWER8 core, PGI compiler, Linux perf profiler)

VII. CONCLUSION

The last stable version of TELEMAC-MASCARET (v8p0r2) is now working on the OPENPOWER architecture with GCC, PGI and with IBM XL (when using a minor patch). An official certification has been granted by the OPENPOWER foundation for this stable version [14].

Progress has been made on porting parts of TOMAWAC to GPUs. This is working as expected and gives performance improvements on subroutines which are computationally intensive such as *qnlin3*, but more work needs to be done for subroutine which are less computationaly intensive, such as *qnlin1*.

ACKNOWLEDGEMENT

This work is supported by the Hartree Centre through the Innovation Return on Research (IROR) programme.

This work was partialy performed using the Cambridge Wilkes service. Part of which is operated by the University of Cambridge Research Computing on behalf of the STFC DiRAC HPC Facility [15]. The DiRAC component of Wilkes was funded by BEIS capital funding via STFC capital grants ST/P002307/1 and ST/R002452/1 and STFC operations grant ST/R00689X/1. DiRAC is part of the National e-Infrastructure.

REFERENCES

[1] Hamza Belaoura, Intégration de la bibliothèque MAGMA dans le système TELEMAC-MASCARET, 2017, Université de Versailles, Saint Quentin en Yvelines, Internship report

[2] https://icl.utk.edu/magma/index.html – Visited on the 2019-08-27

[3] https://openpowerfoundation.org/ – Visited on the 2019-08-27

[4] https://www.nvidia.com/en-gb/data-center/nvlink/ – Visited on the 2019-08-27

[5] https://www.hartree.stfc.ac.uk/Pages/home.aspx – Visited on the 2019-08-27

[6] https://top500.org/system/179044 – Visited on the 2019-08-27

[7] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=87689 – Visited on the 2019-08-27

[8] https://www.pgroup.com/userforum/viewtopic.php?f=4&t=6429 – Visited on the 2019-08-27

[9] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=91410 – Visited on the 2019-08-27

[10] https://www.pgroup.com/userforum/viewtopic.php?f=12&t=6562 – Visited on the 2019-08-27

[11] Sinharoy Balaram, Van Norstrand J. A., Eickemeyer Richard J., et al. IBM POWER8 processor core microarchitecture. IBM Journal of Research and Development, 2015

[12] Moulinec Charles, Denis Christophe, Pham C.-T., et al. TELEMAC: An efficient hydrodynamics suite for massively parallel architectures. Computers & Fluids, 2011, vol. 51, no 1, p. 30-34.

[13] https://perf.wiki.kernel.org/index.php/Main_Page – Visited on the 2019-10-02

[14] https://openpowerfoundation.org/?resource_lib=stfc-daresbury-laboratory-telemac-mascaret-v8 – Visited on the 2019-08-29

[15] www.dirac.ac.uk – Visited on the 2019-09-29