Unified Cross-Platform Profiling of Parallel C++ Applications

Vladyslav Kucher
University of Muenster
Einsteinstr. 62
Muenster, Germany
Email: kucher@wwu.de

Florian Fey
University of Muenster
Einsteinstr. 62
Muenster, Germany
Email: feyf@wwu.de

Sergei Gorlatch
University of Muenster
Einsteinstr. 62
Muenster, Germany
Email: gorlatch@wwu.de

***Abstract*—To address the great variety of available parallel hardware architectures (CPUs, GPUs, etc.), high-performance applications increasingly demand cross-platform portability. While unified programming models like OpenCL or SYCL provide the ultimate portability of code, the profiling of applications in the development process is still done by using different platform-specific tools of the corresponding hardware vendors. We design and implement a unified, cross-platform profiling interface by extending the PACXX framework for unified programming in C++. With our profiling interface, a single tool is used to profile parallel C++ applications across different target platforms. We illustrate and evaluate our uniform profiler using an example application of matrix multiplication for CPU and GPU architectures.**

***Index Terms*—C++, parallelism, many-cores, GPU programming, cross-platform, unified programming model, profiling**

I. Introduction

The great variety of parallel platforms, including CPUs, GPUs and clusters of them from different vendors, makes the cross-platform portability increasingly important for high-performance applications.

While the CUDA [1] programming approach only targets NVIDIA GPU hardware, unified programming models like OpenCL [2], SYCL [3], and PACXX [4] enable applications to run on different hardware architectures without modifying the original source code. However, the existing profiling tools are often closely tailored to the hardware of the respective vendor, so that application developers have to rely on many different tools to ensure the desired portability.

In this work, we suggest to extend the paradigm of unified programming models to unified profiling tools, in order to increase portability in the development process. We aim at profiling high-performance applications using only one cross-platform profiler, rather than requiring a great variety of platform-specific tools. We implement and demonstrate our approach based on the PACXX framework [4] for programming CPU/GPU systems in C++. We extend PACXX with a unified profiling interface that resembles the philosophy of the unified programming model. Our profiling interface seamlessly integrates into the PACXX framework and it allows the collection of profiling information at any given time in the development process, independently of the underlying hardware architecture.

While the portability of code is well studied, less attention has been paid to profiling tools which are mostly tailored to specific hardware architectures. In particular, the existing profiling tools like the NVIDIA Profiler [5], AMD's Radeon Compute Profiler [6] or Intel's VTune Amplifier [7] are proprietary and, thus, do not satisfy the need for portability.

Different approaches have been suggested to increase the flexibility of profiling tools. The flexible profiling interface CUDAAdvisor [8] makes use of the LLVM [9] infrastructure to simultaneously collect performance data in both CPU and GPU portions of CUDA applications. Another experimental approach [10] makes use of CUDA's TAU performance measurement framework to profile GPU applications in high-performance environments. It gives users a detailed feedback about interactions between CPU code and GPU code, without additional modifications to the application's source code. An advancement in the field of profiling GPU applications is the introduction of highly customizable metrics: SASSI (NVIDIA assembly code "SASS" Instrumentor) [11] is a low-level, assembly-language instrumentation tool that enables the definition and injection of arbitrary, user-provided instrumentation code (e.g., placement of custom counters and debugging hooks) to collect fine-grained profiling statistics that would otherwise not be available by default. A generic infrastructure for performance analysis [12] provides a generic interface for performance measurement; it is integrated in the performance infrastructure Score-P to test the performance of OpenCL applications on a variety of hardware architectures.

While previous work has been focused on either improved profiling on a particular GPU architecture, or combined profiling to analyze CPU-GPU interactions, our approach aims at a unified, cross-platform profiling interface that is suited for many different hardware architectures. This should increase usability and allow the collection of profiling information without changing the employed tools whenever the application is ported to another architecture.

The structure of this paper is as follows. In Section II, we present the unified C++ programming model of PACXX and its extension to a unified profiling interface. In Section III, we demonstrate the use of our interface by profiling an example application on different hardware architectures. We summarize our results and conclude in Section IV.

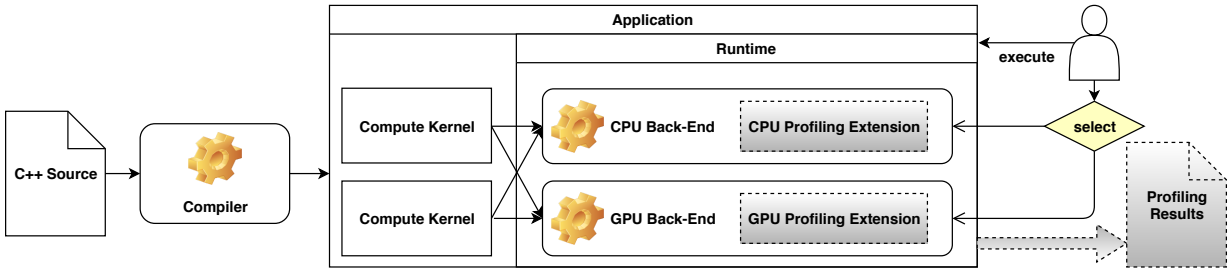

Fig. 1: Structure of the PACXX toolchain, extended with profiling (shaded parts)

II. From unified programming to unified profiling

A. A unified programming model

Our work is based on the PACXX framework [4] that offers a unified parallel programming model based on C++. Similar to the OpenCL standard [13], the PACXX approach is centered around the concept of devices and compute kernels: developers define parallel portions of code as kernels, while the framework provides the required back-end implementations to ensure parallel execution on different device architectures. The major advantage of PACXX is that PACXX applications are single-source C++ code, while in OpenCL compute kernels are written separately from the host code and in a different language.

As shown in Figure 1 (without shaded parts), the PACXX compilation process consists of two separate stages. The first stage is the offline compilation that translates the original source file to a corresponding executable. It precompiles the kernel code and equips the resulting executable with the available back-ends for CPUs and different GPUs. The second stage is the online compilation that takes place when the resulting executable is launched. Since the executable of a compiled PACXX application is automatically equipped with back-ends for each supported hardware architecture, this allows users to select the desired architecture each time the application is executed. The PACXX framework transparently handles the parallel execution on the selected architecture and automatically performs all necessary steps for the actual code generation and optimization. This ensures portability of a PACXX application across different architectures of CPUs and GPUs.

B. Design and implementation of the profiling interface

Figure 1 shows the structure of our unified profiling interface for PACXX applications. We extend the existing PACXX back-ends with individual profiling extensions (shaded parts in the figure) that contain hardware-specific code to capture profiling information. From the user's perspective, our profiling interface serves as a wrapper that automatically selects the appropriate profiling extension depending on the underlying hardware architecture.

Our profiling approach does not require any change to the PACXX offline compiler. The necessary modification is limited to the PACXX runtime and its back-ends that are extended with individual profiling extensions: the CPU back-end makes use of the PAPI library [14], while the back-end for NVIDIA GPUs uses the CUPTI library [15].

C. Structure of the profiling extensions

Our profiling extensions for different back-ends are based on a generic design that relies on a modified kernel launch procedure to manage the collection of performance data. The reason for modification of the kernel launch procedure is that many contemporary devices are incapable of recording different performance metrics simultaneously (e.g., NVIDIA GPUs according to [15]). Depending on the selected metric, they may even require multiple reproducible executions of the same kernel to record a single performance metric. To ensure

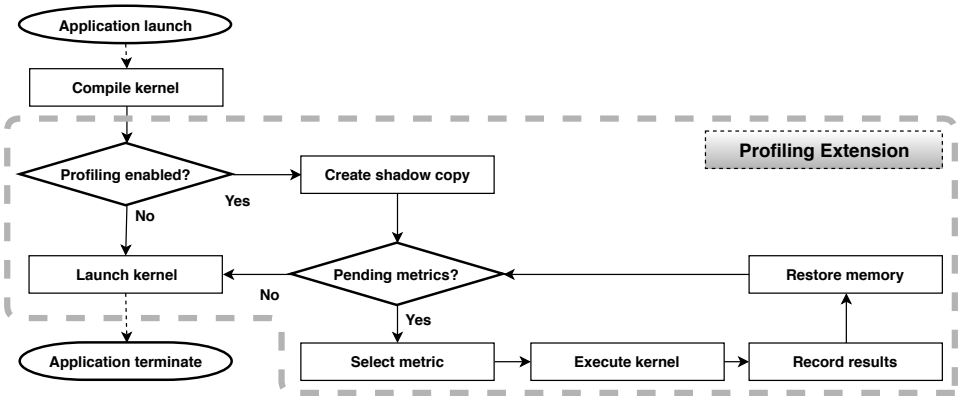

Fig. 2: Kernel launch procedure, modified for profiling

that these executions can be measured independently despite changes of the underlying device memory in every execution, it is necessary to automatically reconstruct the original device memory state after each execution of a kernel. The reconstruction of the original device memory state allows repeated measurements of the same computation, so that accurate performance information is obtained. The standard procedure to achieve this is a shadow copy mechanism that transparently saves and restores the original device memory state whenever the execution of a kernel takes place. This mechanism effectively isolates the measurements from unintended side effects, which eventually ensures precise performance measurements and deterministic behavior across multiple kernel launches.

Figure 2 illustrates our modified launch procedure within an exemplary PACXX back-end, where the added modification is emphasized by the bold box. The modified launch procedure follows the program logic of the profiling extension whose processing steps employ hardware-specific code for a particular back-end. When a kernel is about to be launched, and profiling is enabled, the profiling extension stores a shadow copy of the used device memory. The kernel is then repeatedly executed using for every requested performance metric this shadow copy. Finally, the application proceeds with the actual launch procedure in a regular manner.

III. Use of the profiling extension

We illustrate and evaluate our unified profiling approach using an application example - matrix multiplication.

A. Example application: matrix multiplication

```
1  auto& exec = Executor::get(0);
2
3  auto& dev_a = exec.allocate<double>(matrix_size);
4  auto& dev_b = exec.allocate<double>(matrix_size);
5  auto& dev_c = exec.allocate<double>(matrix_size);
6
7  dev_a.upload(a, matrix_size);
8  dev_b.upload(b, matrix_size);
9  dev_c.upload(c, matrix_size);
10
11 auto pa = dev_a.get();
12 auto pb = dev_b.get();
13 auto pc = dev_c.get();
14
15 auto matMultKernel = [=](auto &config)
16 {
17    auto column = config.get_global(0);
18    auto row    = config.get_global(1);
19    double val  = 0;
20    for (unsigned i = 0; i < width; ++i)
21       val += pa[row * width + i] *
                pb[i * width + column];
22    pc[row * width + column] = val;
23 };
24 exec.launch(matMultKernel,
     {{width/threads , width} , {threads , 1} , 0});
25
26 dev_c.download(c, matrix_size);
```

Fig. 3: The matrix multiplication benchmark in PACXX

Figure 3 shows the matrix multiplication example program in PACXX, taken from [16], where the PACXX code is derived from the sequential C++ code by expressing nested loops as parallel kernel calls. Further, the program is extended with code for memory management to transfer inputs and outputs between the involved architectures. Arrays `a` and `b` represent the two input matrices, the result of the matrix multiplication is stored in array `c`. The `matMultKernel` compute kernel (lines 15-23) is represented by a C++ lambda expression that will be executed in parallel (once for every element of the result matrix). Similar to CUDA and OpenCL, the execution of a compute kernel requires some manual arrangement by uploading the required input data (line 3-13) and fetching the corresponding result (line 26) once the computation is finished.

Our example code in Figure 3 uses the canonical double-layered, two-dimensional parallelization scheme by partitioning into blocks and threads. The `launch` function (line 24) starts the execution of the kernel and defines the distribution of work by defining a block-level and thread-level parallelization in up to three dimensions. This determines the amount and structure of the exploited parallelism. The first component of the range declares the overall number of started blocks, while the `threads` variable in the second component of the range declares the number of threads per block. During the execution, every block is scheduled to one processor that processes all threads of the block according to the SIMT (Single Instruction, Multiple Threads) paradigm - a combination of SIMD (Single Instruction, Multiple Data) and multithreading paradigms. The `get_global` function is part of the kernel configuration; it allows the retrieval of the corresponding thread id in every dimension of the range. In the example, the first dimension of the range parameter corresponds to the row while the second component indicates the column of the processed element in the matrix. The PACXX offline compiler is invoked by the command line:

```
> pacxx++ -o matMult matMult.cpp
```

The build process automatically equips the resulting executable with our unified profiling interface. The PACXX offline compiler behaves as a wrapper for the standard Clang compiler [16], so that the PACXX framework effectively serves as a drop-in replacement for the existing C++ LLVM compilation toolchain.

B. Profiling the matrix multiplication benchmark

Figure 4 shows how the profiling of our example application is started by invoking the executable with the *PACXX_PROF_ENABLE* runtime environment variable enabled. This instructs the PACXX runtime environment to automatically profile every kernel instance upon its invocation. Whenever a kernel is executed, the runtime environment performs the online compilation stage transparently to the user and enables the profiling extension, so that the profiling information is gathered from the underlying device. The *PACXX_DEFAULT_RT* environment variable determines the

used architecture, which can be either a *CPU* or a *GPU*. The resulting data is written to an output file that can be explicitly specified by an environment variable as shown in Figure 4. If no output file is specified, the profiling information is displayed on the standard output. By default only one metric - the execution time - is recorded.

```
> PACXX_DEFAULT_RT=GPU
  PACXX_PROF_ENABLE=1
  PACXX_PROF_IN=profiling_configuration
  PACXX_PROF_OUT=results.json
  ./matMult
```

Fig. 4: Envoking a PACXX application with profiling

The user can specify a custom configuration file, for example *cf_executed*, that serves as the argument for the *PACXX_PROF_IN* enviroment variable to request for additional performance metrics. Our profiler can record all metrics provided by the corresponding architecture's vendor, e.g., memory throughput, instructions per cycle, memory utilization, etc. A list of all 174 available performance metrics for NVIDIA GPUs can be found in [15].

C. Visualization of profiling results

Figure 5 shows the results of profiling started as in Figure 4 for the matrix multiplication benchmark. The results are presented as a well-structured, human-readable JSON file that contains the requested performance metrics for every launched kernel instance. When multiple launches of a kernel occur, the profiler accordingly tracks every single kernel launch independently: the individual profiling results in the output file are then prepended by the function name of the corresponding kernel. This gives the user an instant feedback about the behaviour of the performed computation, without manually selecting the appropriate profiling tool for each individual kernel and the hardware on which the kernel is executed: the PACXX runtime environment automatically ensures the use of the appropriate profiling functionality, depending on the underlying hardware architecture.

```
{
  "matMultKernel": [
    {
      "Metrics": {
        "cf_executed": "68157440",
        "kernelDuration": "517349164ns"
      }
    }
  ]
}
```

Fig. 5: The profiling results in a human-readable JSON format

Due to the standardized JSON output of our profiling interface, it is easy to visualize the profiling results with plotting tools like Gnuplot. This allows a convenient comparison of the application's behavior across different hardware architectures.

For illustration, we evaluate the matrix multiplication benchmark on a CPU (Intel Xeon E5-1620 v2) and on a GPU (NVIDIA Tesla K20c). The input data of the benchmark consists of two randomly generated dense 4096x4096 square matrices. Figure 6 shows a comparison of the benchmark's execution time on both architectures: the diagram shows how the execution time depends on the amount of used parallelism that is determined by the `threads` parameter in the code (see Figure 3).

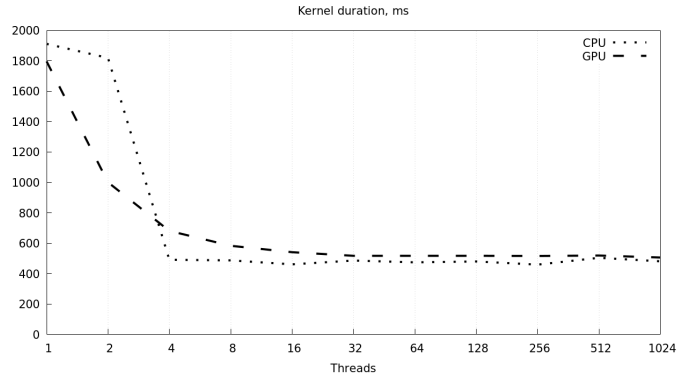

Fig. 6: Execution time of the matrix multiplication benchmark (CPU/GPU comparison)

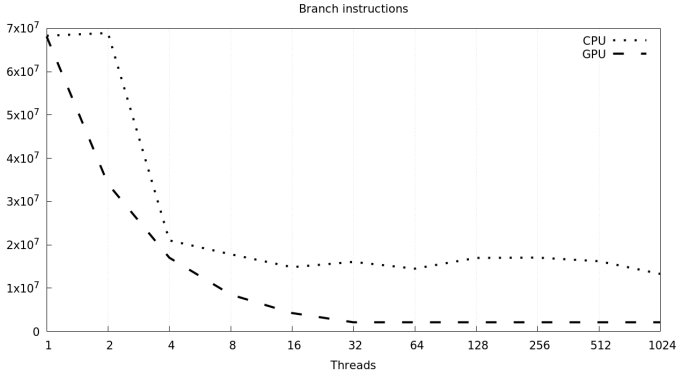

Fig. 7: Number of branch instructions for the matrix multiplication benchmark (CPU/GPU comparison)

Besides the execution time, there are additional metrics that allow for a more detailed comparison of different hardware architectures. Figure 7 compares the number of branch instructions on both the CPU and the GPU depending on the `threads` parameter (see Figure 3).

With profiling, it is possible to visualize different properties of an application dependent on certain parameters. This allows a test-driven selection of optimal parameter configurations that satisfy the desired properties. Our profiling approach improves on this concept by providing a uniform specification of profiling metrics and uniform representation of profiling results,

so that different hardware architectures can be conveniently compared to each other.

IV. Conclusion

Proprietary programming and profiling frameworks like CUDA provide powerful tools for hardware-accelerated parallel programming, but they are not portable. The proposed portable frameworks for parallel cross-platform applications - OpenCL [2], SYCL [3] and PACXX [4] - target multiple hardware architectures, including CPUs and GPUs, but they do not offer cross-platform profiling.

In this paper, we describe the design and implementation of our unified profiling interface. Its main advantage compared to the existing profiling approaches is the extensive support for a hardware-independent profiling of cross-platform applications. Our approach avoids the necessity to use several tools in order to profile the same application on different hardware architectures. The unified profiling approach increases the flexibility of the development process: the application itself can be configured to produce profiling information on every supported hardware architecture, such that the use of additional development tools becomes obsolete. We provide a convenient interface that seamlessly integrates into the PACXX programming framework without any modification of existing applications.

Acknowledgment

This work was generously funded by the German Federal Ministry of Education and research (BMBF) in the HPC^{2}SE project.

References

[1] J. Nickolls, I. Buck, M. Garland and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[2] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "OpenCL As a Unified Programming Model for Heterogeneous CPU/GPU Clusters," *SIGPLAN Not.*, vol. 47, no. 8, pp. 299–300, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2370036.2145863

[3] R. Keryell, R. Reyes and L. Howes, "Khronos SYCL for OpenCL: A Tutorial," in *Proceedings of the 3rd International Workshop on OpenCL*, ser. IWOCL '15. New York, NY, USA: ACM, 2015, pp. 24:1–24:1. [Online]. Available: http://doi.acm.org/10.1145/2791321.2791345

[4] M. Haidl and S. Gorlatch, "PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14," in *2014 LLVM Compiler Infrastructure in HPC (SC'14)*, Nov 2014, pp. 1–11.

[5] "NVIDIA Visual Profiler," https://developer.nvidia.com/nvidia-visual-profiler, accessed: 2018-07-14.

[6] "Radeon Compute Profiler," https://github.com/GPUOpen-Tools/RCP, accessed: 2018-07-14.

[7] "Intel VTune Amplifier," https://software.intel.com/intel-vtune-amplifier-xe, accessed: 2018-07-14.

[8] D. Shen, S. L. Song, A. Li and X. Liu, "CUDAAdvisor: LLVM-based Runtime Profiling for Modern GPUs," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: ACM, 2018, pp. 214–227. [Online]. Available: http://doi.acm.org/10.1145/3168831

[9] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[10] A. D. Malony, S. Biersdorff, W. Spear and S. Mayanglambam, "An Experimental Approach to Performance Measurement of Heterogeneous Parallel Applications Using CUDA," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 127–136. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810105

[11] M. Stephenson, S. Hari, S. Kumar, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, S. W. Keckler, "Flexible Software Profiling of GPU Architectures," *SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 185–197, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2872887.2750375

[12] R. Dietrich and R. Tschter, "A generic infrastructure for OpenCL performance analysis," in *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1, Sept 2015, pp. 334–341.

[13] J. E. Stone, D. Gohara and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2010.69

[14] J. Dongarra, K. London, S. Moore, P. Mucci and D. Terpstra, "Using PAPI for Hardware Performance Monitoring on Linux Systems," 08 2009.

[15] "CUPTI API," https://docs.nvidia.com/cuda/cupti/r_main.html, accessed: 2018-07-14.

[16] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD conference*, 2008, pp. 1–2.

V. Artifact description: "Unified Cross-Platform Profiling of Parallel C++ Applications"

A. Abstract

This artifact describes the experiments that were conducted for the paper "Unified cross-platform profiling of parallel applications in C++". The following workflow guides the reviewer from installing software dependencies and compiling the experiments source code to obtaining the results.

B. Description

1) How software can be obtained: The artifact is hosted at https://zivgitlab.uni-muenster.de/HPC2SE-Project/pacxx

2) Check-list (artifact meta-information):

- **Program:** The PACXX Framework and the matrix multiplication sample benchmark implemented in it.
- **Compilation:** During workflow.
- **Run-time environment:** Any computer with Ubuntu 16.04.5.
- **Hardware:** Intel Xeon E5-1620 v2 CPU with at least 16GBs of RAM and NVIDIA Tesla K20c GPU.
- **Output:** Filename specified by a runtime environment variable or displayed on the standard output.
- **Experiment workflow:** Obtain artifact; Obtain PACXX Framework; Configure PACXX Framework; Compile and install PACXX Framework; Obtain sample benchmark; Configure sample benchmark; Compile sample benchmark with PACXX Framework; Run sample benchmark to obtain results;
- **Experiment customization:** Via PACXX.prof or a custom filename specified by a runtime environment variable.
- **Publicly available?:** Yes

3) Hardware requirements: Intel Xeon E5-1620 v2 CPU with at least 16GBs of RAM and NVIDIA Tesla K20c GPU.

4) Software dependencies:

- Python 2.7 and 3.x
- a compiler supporting C++14 or higher
- CMake 3.5 or higher
- Intel TBB library
- PAPI library
- Ninja build system
- CUDA runtime

C. Preparing the artifact

The installation process of the artifact consists in obtaining, configuring, compiling and installing PACXX Framework as well as obtaining, configuring and compiling the sample benchmark.

1) Obtaining PACXX Framework: ./getpacxx

2) Configuring PACXX Framework: ./preppacxx

3) Compiling and installing PACXX Framework: ./build-pacxx

4) Obtaining the sample benchmark: ./getsamples

5) Configuring the sample benchmark: ./prepsamples

6) Building the sample benchmark: cd bsamples && ninja matmul

D. Experiment workflow

The reviewer is invited to perform the following steps:

1) Create a file named *PACXX.prof* with the following contents:

```
PAPI_TOT_INS
PAPI_LD_INS
PAPI_SR_INS
PAPI_BR_INS
inst_executed
gld_transactions
gst_transactions
cf_executed
```

2) Perform profiling on the CPU using *PACXX_DEFAULT_RT=1 PACXX_PROF_ENABLE=1 PACXX_PROF_OUT=CPU.json ./matmul/matmul*.
3) Perform profiling on the GPU using *PACXX_DEFAULT_RT=0 PACXX_PROF_ENABLE=1 PACXX_PROF_OUT=GPU.json ./matmul/matmul*.
4) Prepare the profiling results for plotting using the python3 scripts provided under https://zivgitlab.uni-muenster.de/HPC2SE-Project/pacxx/tree/master/PMBS18:
   a) If you opted in for 3 use *Both-plotter.py*.
   b) Otherwise use *CPU-plotter.py*.
5) Generate the plots using *gnuplot5 plot.gnu*.

E. Evaluation and expected results

We evaluate the matrix multiplication benchmark provided in Figure 3. We expect the evaluation results that are analogous to those presented in our experimental results in Figures 6-7.