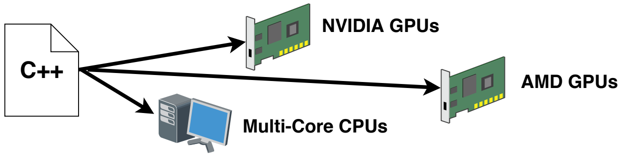
Unified Cross-Platform Profiling of Parallel C++ Applications

Vladyslav Kucher   Florian Fey   Sergei Gorlatch

University of Muenster, Germany

PMBS' 2018

Motivation: Cross-Platform HPC Applications



HPC demands cross-platform portability to

- reuse existing code on current and future hardware;
- remain independent of specific hardware vendors;
- conveniently develop applications based on unified interfaces.

Challenge: Profilers are usually hardware-specific.

Our focus: unified, hardware-independent profiling.

State-of-the-Art Profiling

Profiling tools are usually tailored to specific architectures:

- NVIDIA Profiler for NVIDIA GPUs
- AMD's Radeon Compute Profiler for AMD GPUs
- Intel's VTune Amplifier for Intel CPUs

Existing proprietary profilers are not portable.
Few generic performance analysis tools have been suggested.

Unified Programming in PACXX [Uni Muenster]

PACXX is a framework for HPC programming in C++:

- Parallel programming using C++ with no language extensions.
- Advanced C++14 features: Lambdas, Templates, STL.
- Unified programming model: no separation of host vs. device.
- All kinds of accelerators: NVIDIA/AMD GPUs and CPUs
- Automatic memory handling and GPU configuration.
- Advanced optimizations: multi-stage programming, tiling, etc.

Example: Matrix Multiplication in C++ vs. PACXX

C++

```
double a[size]; double b[size]; double c[size];             // matrices
for (int row = 0 ; row < width ; ++row)                     // iterate over rows
   for (int column = 0 ; column < width ; ++column)         // iterate over columns
   {
      double val  = 0;                                      // process element
       for (unsigned i = 0; i < width; ++i)                 //
          val += a[row * width + i] * b[i * width + column]; //
       c[row * width + column] = val;                       //
   }
```

PACXX

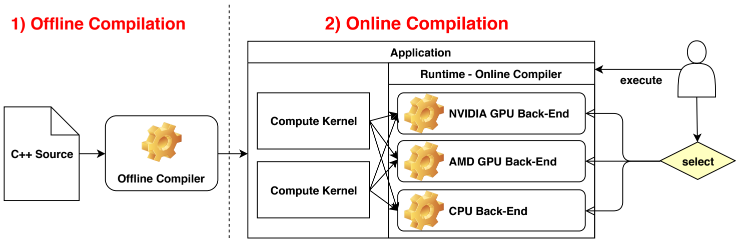
```
auto [a, b, c] = managed_ref<vector, double>(size, size, size); // matrices
pacxx::launch([=](auto &config)                                 // kernel lambda
{                                                               //
   auto column = config.get_global(0);                          // index column
   auto row    = config.get_global(1);                          // index row
   double val  = 0;                                             // process element
   for (unsigned i = 0; i < width; ++i)                         //
      val += a[row * width + i] * b[i * width + column];        //
   c[row * width + column] = val;                               //
},{{width/threads, width},      {threads, 1}      });           // execute the kernel
```

$$\underbrace{\{width/threads, width\}}_{\text{task parallelism}}, \quad \underbrace{\{threads, 1\}}_{\text{data parallelism (vectorization)}}$$

Easy porting C++ $\rightarrow$ PACXX.

PACXX code is much shorter compared to CUDA and OpenCL!

Toolchain of the PACXX Framework

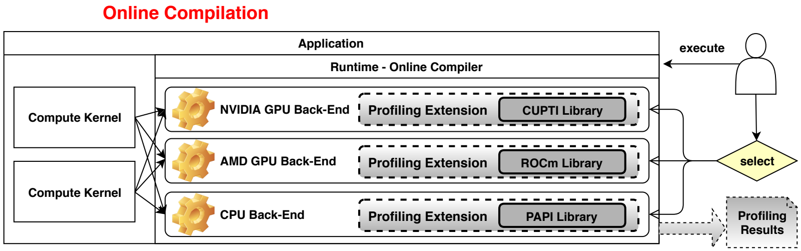


PACXX is implemented as a two-stage C++ compiler:

1. Offline compilation translates C++ source code and outputs an executable containing a custom runtime.
2. Online compilation uses specialized back-ends on GPUs (NVIDIA and AMD) and CPUs from different vendors.

Target architecture is selected at runtime.
Future targets require no change of existing applications!

Extending PACXX with Unified Profiling



We extend back-ends of PACXX with profiling extensions, built on top of vendor-specific profiling libraries:

- the PAPI library for CPUs;
- the CUPTI library for NVIDIA GPUs;
- the ROC library for AMD GPUs (work in progress).

Our implementation only changes the PACXX runtime (shaded), no change of the PACXX offline compiler is required!

Profiling extensions are accessible by a unified interface.

The Interface for Unified Profiling

Starting the application with profiling enabled:

```
> PACXX_DEFAULT_RT=GPU_NVIDIA     # selects the NVIDIA GPU
  PACXX_PROF_ENABLE=1             # enables profiling
  PACXX_PROF_IN=metrics.conf      # configure metrics by file
  PACXX_PROF_OUT=results.json     # output file
  ./matMult
```

Output file produced by the profiling interface in JSON:

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

Metrics shown here: control flow instructions and execution time.

Implementation of Profiling Extensions: Challenge

The hardware provides only few debug registers and counters.
Profiling complex metrics (e.g., branches) requires more data.
$\Rightarrow$ Multiple measurements of a kernel launch are required

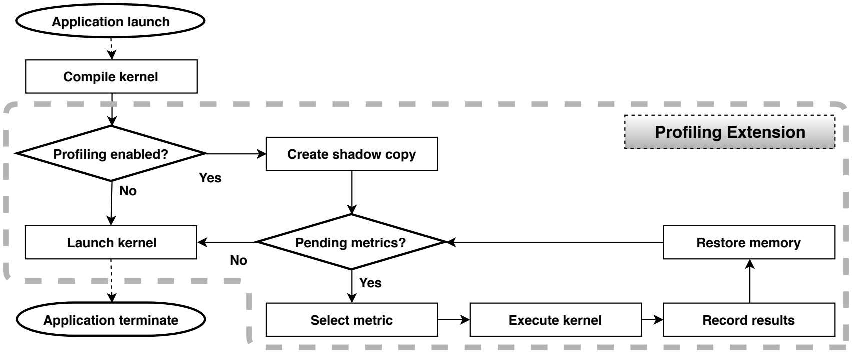
Problem: Repeated launches of a kernel are not reproducible, because kernels change memory which leads to side effects!

```
int memory[] = {0,1,2,3,4,5,6,7};

auto kernel = [=](auto &config)
{
   int i = config.get_global(0);
   if (memory[i] == 42)
   {
      memory[i] = 7;   // second launch: memory = {7,7,...}
   }
   else
   {
      memory[i] = 42;  // first launch: memory = {42,42,...}
   }
};
```

Implementation of Profiling Extensions: Solution

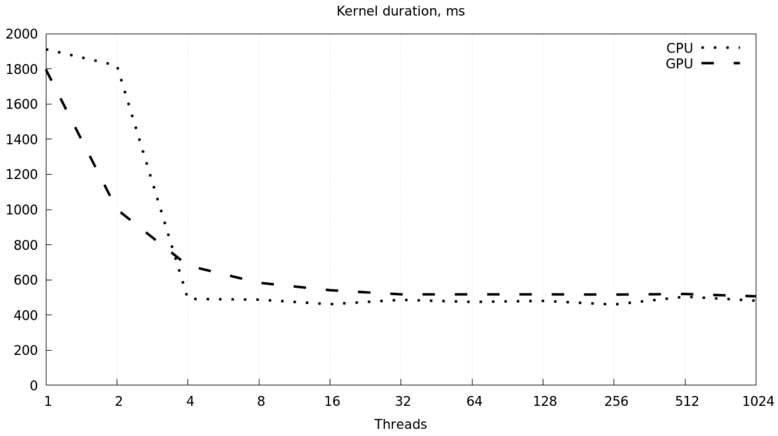
Solution: Eliminate side effects to measure kernels repeatedly, by implementing a **modified kernel launch procedure**.



Shadow copy mechanism transparently saves/restores memory, and repeatedly runs kernel code until all metrics are recorded.

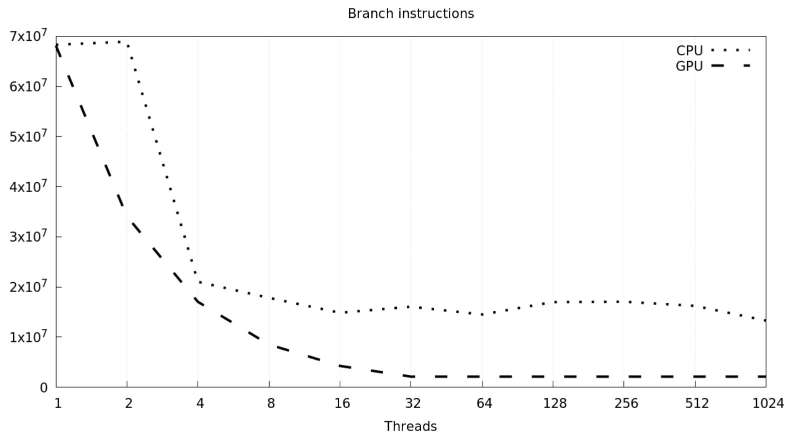
⇒ Precise measurement of complex profiling metrics, e.g. branches

Visualization of Profiling Results (Execution Time)



NVIDIA Tesla K20c GPU vs. Intel Xeon E5-1620v2 CPU (4 cores).
CPU: Improvement with 4 threads due to **vectorization**.
GPU: Continuous improvement up to 32 threads (warp size).

Visualization of Profiling Results (Branch Instructions)



Similar branch behavior on CPU/GPU, comparable performance.
CPU: Notable change for 4 threads due to **vectorization**.
GPU: Decrease of branching up to 32 threads (warp size).

Conclusion

We develop and implement a unified profiling interface for:

- hardware-independent profiling without additional tools;
- unified workflow in developing C++ HPC applications.

Our modular design with different back-ends:

- ensures portability to current and future hardware;
- abstracts over hardware-specific profiling tools.

We support CPUs from different vendors; GPUs from NVIDIA; GPUs from AMD (work in progress).

Questions?