2021 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools) | 978-1-6654-1110-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/ProTools54808.2021.00009

Measurement and Analysis of GPU-Accelerated OpenCL Computations on Intel GPUs

Aaron Thomas Cherian, Keren Zhou, Dejan Grubisic, Xiaozhu Meng, John Mellor-Crummey
Dept. of Computer Science
Rice University
{atc8,kz21,dx4,xm13,johnmc}@rice.edu

***Abstract*—Graphics Processing Units (GPUs) have become a key technology for accelerating node performance in supercomputers, including the US Department of Energy's forthcoming exascale systems. Since the execution model for GPUs differs from that for conventional processors, applications need to be rewritten to exploit GPU parallelism. Performance tools are needed for such GPU-accelerated systems to help developers assess how well applications offload computation onto GPUs.**

In this paper, we describe extensions to Rice University's *HPCToolkit* performance tools that support measurement and analysis of Intel's *DPC++* programming model for GPU-accelerated systems atop an implementation of the industry-standard *OpenCL* framework for heterogeneous parallelism on Intel GPUs. *HPCToolkit* supports three techniques for performance analysis of programs atop *OpenCL* on Intel GPUs. First, *HPCToolkit* supports profiling and tracing of *OpenCL* kernels. Second, *HPCToolkit* supports *CPU-GPU blame shifting* for *OpenCL* kernel executions—a profiling technique that can identify code that executes on one or more CPUs while GPUs are idle. Third, *HPCToolkit* supports fine-grained measurement, analysis, and attribution of performance metrics to *OpenCL* GPU kernels, including instruction counts, execution latency, and SIMD waste. The paper describes these capabilities and then illustrates their application in case studies with two applications that offload computations onto Intel GPUs.

***Index Terms*—Supercomputers, High performance computing, Performance analysis, Parallel programming**

I. Introduction

Supercomputers composed of compute nodes accelerated with Graphics Processing Units (GPUs) are becoming increasingly common. GPUs are a popular kind of accelerator because they provide enormous computation rates with much better power efficiency than traditional CPUs. For that reason, all of the forthcoming exascale computing platforms being developed by the US Department of Energy employ GPU-accelerated compute nodes.

The Aurora exascale supercomputer, being developed by Intel for Argonne National Laboratory, will be composed of compute nodes equipped with two Intel Xeon "Sapphire Rapids" CPUs and six Intel Xe "Ponte Vecchio" GPUs [1]. While NVIDIA's ecosystem for *CUDA* is well-established, developing a capable software ecosystem for Intel's emerging GPU-accelerated compute nodes is a work in progress. At present, many parts of the Aurora hardware and software are undergoing rapid development, including Intel's forthcoming Ponte Vecchio GPUs, Intel's compilers for generating GPU code, Intel's *OpenCL* and *Level Zero* runtime systems for heterogeneous parallel computing, as well as tools for performance measurement, analysis, and modeling.

As part of the Exascale Computing Project, Rice University is extending its *HPCToolkit* performance tools [2], [3] for measurement and analysis of GPU-accelerated applications on compute nodes accelerated with Intel GPUs. *HPCToolkit* is a suite of tools for measurement and analysis of the performance of highly-optimized programs on systems ranging from desktops to supercomputers. *HPCToolkit* supports profiling within and across nodes of a parallel system. Its principal mode of performance measurement on CPUs is asynchronous sampling, which leads to low measurement overhead with appropriately chosen sampling frequencies. *HPCToolkit* can measure a wide range of performance metrics including time as well as hardware counter metrics, which include measures of work (e.g. graduated instructions) and inefficiency (e.g. cache misses, branch mis-predictions, or stalls). In a post-mortem analysis phase, *HPCToolkit* correlates metrics with static and dynamic contexts in application source code.

In prior work, we describe extensions to *HPCToolkit* for measurement and analysis of GPU-accelerated programs on compute nodes accelerated with NVIDIA GPUs [3]. Here, we describe new features added to *HPCToolkit* to profile code offloaded to Intel GPUs using the *OpenCL* framework for heterogeneous parallel computing. New capabilities developed to support monitoring of computations offloaded to Intel GPUs using *OpenCL* include:

- support for collecting call path profiles and call path traces of *OpenCL* operations including kernel computations, data copies, and synchronizations,
- support for *CPU-GPU blame shifting* to quantitatively assess opportunity costs associated with code regions where no *OpenCL* kernel is active on GPUs, and
- support for collecting, analyzing, and attributing fine-grain measurements within kernels on Intel GPUs.

HPCToolkit attributes measurements of *OpenCL* operations to the full call path in the thread that initiated them. *HPCToolkit* attributes fine-grain measurements of computations performed by an *OpenCL* kernel to source lines and loops within the kernel.

Section II introduces background material for the paper. Section III outlines additions to *HPCToolkit* for profiling and tracing of *OpenCL* operations. Section IV describes measurement strategies for identifying opportunities to improve application performance. Section V describes *HPCToolkit's* implementation of several capabilities for fine-grain measurement and analysis of kernels executing on Intel GPUs. Section VI describes case studies that illustrate the utility of the capabilities described in this paper. Section VII provides an overview of related work on GPU performance measurement. Section VIII summarizes our conclusions and outlines our plans to extend this work in the future.

II. Background

A. Intel GPU hardware

HPCToolkit's support for Intel GPUs was developed using an Intel GEN9 GPU as a proxy for Intel's forthcoming Ponte Vecchio GPU. The principal computational element of a GEN9 GPU is known as an *Execution Unit (EU)*. Each EU has a pair of Single Instruction Multiple Data (SIMD) Arithmetic Logic Units (ALU), seven register sets—one for each hardware thread, thread control logic, a branch unit, and a send unit that manages data movement in and out of the EU. A *SubSlice* consists of a group of eight EUs along with a shared local memory and a memory interface known as a data port. A *Slice* consists of three SubSlices along with an L3 cache.

Intel's forthcoming Ponte Vecchio GPU [4], which will serve as the foundation of compute nodes in Aurora, has a similar hierarchical organization. The principal computational element is a X^e *Core*, which has eight vector engines for executing SIMD instructions, eight matrix engines (similar to NVIDIA's tensor cores), register sets for thread contexts, as well as instruction and L1 data cache/shared local memory. 16 X^e cores are organized into *Slices*. A *Stack* consists of up to four Slices, an L2 cache, memory controllers, and communication links. A *2-Stack* consists of a pair of Stacks and finally a full GPU may have as many as eight 2-Stacks.

At the heart of Intel's GPU designs are functional units for executing SIMD instructions that operate on multiple data elements at a time. AMD's GPUs also employ SIMD units while NVIDIA's GPUs do not. On GPUs that support SIMD execution, analyzing the utilization of SIMD lanes is important for assessing the degree to which a program is fully exploiting the massive vector parallelism of the GPU.

B. SIMD inefficiencies

a) SIMD divergence: SIMD divergence occurs in if-else blocks of GPU kernels. GPUs use SIMD instructions that require execution of the same operation on all SIMD lanes of a hardware thread. Some lanes may need to execute the *if* block while others may need to execute the *else* block. But because the same instruction must execute in all lanes, the runtime will execute each if block and each else block but with complementary lane masks for each. If a lane's mask is false, it will not perform the operation.

b) SIMD waste: Each SIMD instruction has multiple lanes—one for each data element. SIMD waste occurs when all lanes do not contribute to a computation. Primary factors that cause SIMD waste are SIMD divergence, address arithmetic, uniform variables (constant values for all SIMD lanes), reduction operations, and message payloads.

Code transformations can help reduce/eliminate SIMD waste. The Intel optimization guide [5] describes several methods to avoid conditional checks, such as padding buffers and replacing conditional checks by relational functions. Address arithmetic associated with arrays can be reduced by manually hoisting accesses out of loops or performing transformations such as scalar replacement [6] if the compiler doesn't.

C. CPU-GPU Blame Shifting

CPU-GPU blame shifting [7] is a measurement and analysis strategy that aims to identify code regions in GPU-accelerated applications that cause idleness. This strategy can help identify CPU code on the critical path of an application execution where GPUs are idle, which represent potential opportunities for improving performance by offloading.

D. OpenCL *Execution Model [8]*

A host runs an *OpenCL* application that offloads work to other compute devices. In *OpenCL* jargon, a *context* refers to the environment that includes the devices, memory objects, the link between the host and the device (queue), etc. On an *OpenCL* device, a work item is the fundamental unit of work and a kernel is the function applied to a work item. Work-items are grouped into work-groups.

E. Data parallel C++

Data parallel C++, known as *DPC++* or *DPCPP*, is a high level language developed by Intel as an extension of the industry-standard *SYCL* programming model [9] for heterogeneous platforms. *DPC++* can be compiled to either Intel's *OpenCL* or *Level Zero* runtime system.

F. Latency hiding

GPUs use multi-threading to hide stalls within a thread's instruction stream. GPU instruction stalls come from a variety of causes, including execution pipeline latency, load latency, and branch delays. A thread is marked ready-to-run if all registers used by its next instruction are ready. Whenever a thread experiences a stall, the EU switches to the next ready-to-run thread. The more ready-to-run threads an EU has, the better the EU can hide the latency of stalled threads. The formula for calculating the number of hardware threads needed in the EU to hide latency is as follows: Let covered latency (C) be the number of cycles that the functional units take for instruction execution. Let uncovered latency (U) be the number of cycles lost in stalls. The minimum hardware threads t needed by an EU to cover latency is $t = 1 + (U/C)$ [10].

However, the number of threads in an EU is limited by the number of register sets. If many threads are needed to hide latency, it could be a sign that your thread's workload or access patterns need to be adjusted.

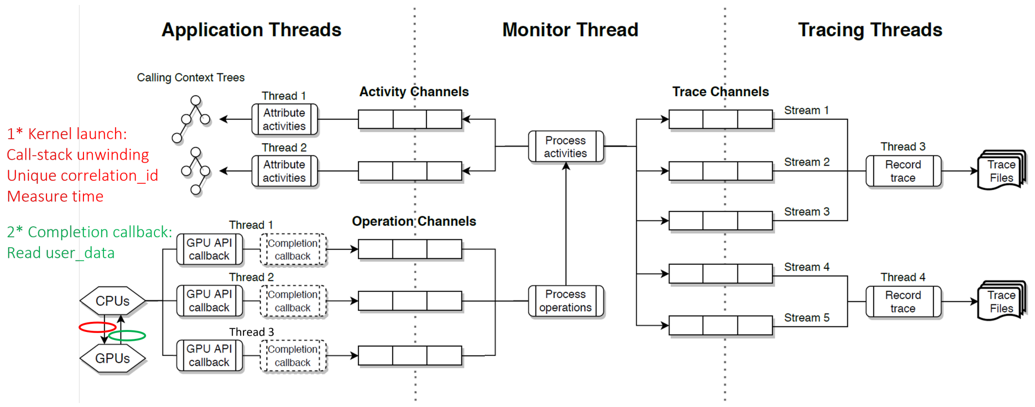
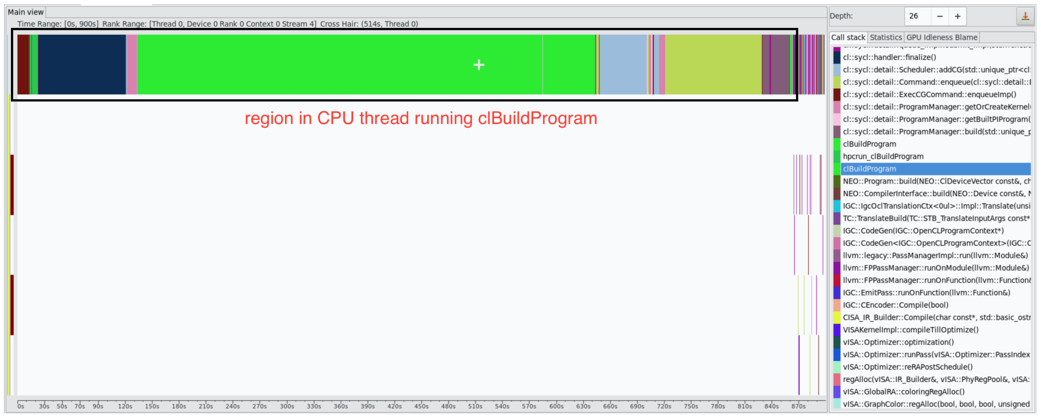


Fig. 1. *HPCToolkit*'s tracing infrastructure coordinating application threads, monitor thread, and tracing threads.


Main view
Time Range: [0s, 900s] Rank Range: [Thread 0, Device 0 Rank 0 Context 0 Stream 4] Cross Hair: (514s, Thread 0)
region in CPU thread running clBuildProgram
Depth: 26
Call stack
Statistics
GPU Idleness Blame

```
cl::sycl::handler::finalize()
cl::sycl::detail::Scheduler::addCG(std::unique_ptr<cl
cl::sycl::detail::Command::enqueue(cl::sycl::detail::
cl::sycl::detail::ExecCGCommand::enqueueImp()
cl::sycl::detail::ProgramManager::getOrCreateKernel
cl::sycl::detail::ProgramManager::getBuiltPIProgram(
cl::sycl::detail::ProgramManager::build(std::unique_p
clBuildProgram
hpcrun_clBuildProgram
clBuildProgram
NEO::Program::build(NEO::ClDeviceVector const&, ch
NEO::CompilerInterface::build(NEO::Device const&, N
IGC::IgcOclTranslationCtx<0ul>::Impl::Translate(unsi
TC::TranslateBuild(TC::STB_TranslateInputArgs const*
IGC::CodeGen(IGC::OpenCLProgramContext*)
IGC::CodeGen<IGC::OpenCLProgramContext>(IGC::C
llvm::legacy::PassManagerImpl::run(llvm::Module&)
llvm::FPPassManager::runOnModule(llvm::Module&)
llvm::FPPassManager::runOnFunction(llvm::Function&
IGC::EmitPass::runOnFunction(llvm::Function&)
IGC::CEncoder::Compile(bool)
CISA_IR_Builder::Compile(char const*, std::basic_ostr
VISAKernelImpl::compileTillOptimize()
vISA::Optimizer::optimization()
vISA::Optimizer::runPass(vISA::Optimizer::PassIndex
vISA::Optimizer::reRAPostSchedule()
regAlloc(vISA::IR_Builder&, vISA::PhyRegPool&, vISA:
vISA::GlobalRA::coloringRegAlloc()
vISA::GraphColor::regAlloc(bool, bool, bool, unsigned
```

Fig. 2. Trace view for application *Amr-Wind* shows the active and idle times for CPU threads and GPU streams [7]

III. Profiling and Tracing *OpenCL* Operations

HPCToolkit collects GPU profiles and traces data in three steps. In the first step, it intercepts GPU invocation launches and unwinds CPU call stacks to identify calling contexts. Then it collects and processes GPU performance metrics associated with invocations (step 2) and attributes them back to CPU calling contexts (step 3). These three steps are achieved with the interaction between application threads, monitor thread, and tracing threads shown in Figure 1. Threads exchange data through bidirectional single-producer single-consumer and wait-free channels described in [3]. Each application thread is responsible for unwinding its CPU call stacks and attributing GPU metrics to its own calling contexts (steps 1 and 3). The monitor thread is responsible for collecting GPU activities from the profiling APIs, processing the measurements and sending data to the application, and tracing threads for recording.

For *OpenCL* applications, depending upon the GPU operation invoked, either an application thread or a runtime thread will receive a completion callback providing measurement data. At each GPU API invocation $\mathcal{I}$ by an application thread $\mathcal{T}$, *hpcrun* provides a `user_data` parameter, which includes a placeholder node $\mathcal{P}$ for the invocation $\mathcal{I}$ and $\mathcal{T}$'s activity channel $\mathcal{C}_{\mathcal{A}}$. The *OpenCL* runtime will pass `user_data` to the completion callback associated with $\mathcal{I}$.

At each completion callback, some threads receive measurement data about a GPU activity $\mathcal{A}$. Using information from its `user_data` argument, the completion callback correlates $\mathcal{A}$ with placeholder $\mathcal{P}$ and then en-queues an operation of $(\mathcal{A}, \mathcal{P}, \mathcal{C}_{\mathcal{A}})$ for the monitor thread in its operation channel $\mathcal{C}_{\mathcal{O}}$. The monitor thread en-queues an $(\mathcal{A}, \mathcal{P})$ pair in $\mathcal{T}$'s activity channel $\mathcal{C}_{\mathcal{A}}$. This way each application thread will correlate the measurement data from an GPU operation it launched to the invocation calling context previously collected.

IV. Analyzing Application Performance

A. *CPU-GPU Blame-Shifting for* OpenCL

CPU-GPU blame shifting aims at identifying work along the critical path of an *OpenCL* application's execution. Of particular interest is CPU code along the critical path where the GPU is idle. *HPCToolkit's* CPU-GPU blame-shifting implementation for *DPC++* is based on prior work by Chabbi et al. [7]. That work wrapped *CUDA* entry points to monitor GPU operations. Ours is a more generic approach that relies on

TABLE I
Amr-Wind CPU FUNCTIONS CONTRIBUTING MOST TO GPU IDLENESS

CPU function	GPU idleness (%)
incflo::InitialProjection	60.07
incflo::InitialIterations	24.46
incflo::init_amr_wind_modules	12.17

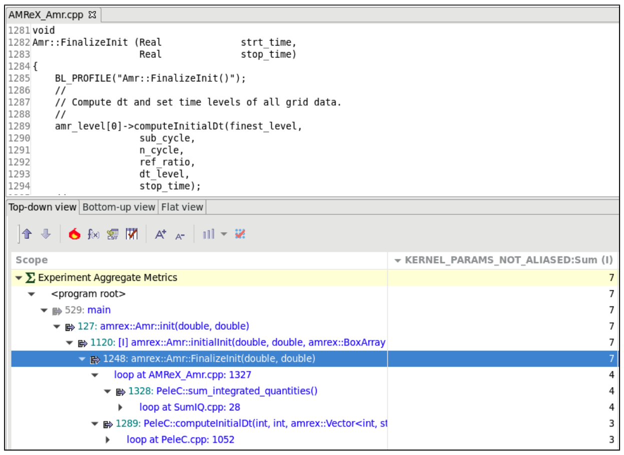
intercepting *OpenCL* operations. Supporting blame shifting for *OpenCL* implementation layer for *DPC++* naturally extends the support to *DPC++*. With some additional generalization, this implementation could be used to build support for other GPU programming models, including *HIP* and *CUDA*.

HPCToolkit calculates three basic blame-shifting metrics: CPU_IDLE (sec), CPU_IDLE_CAUSE (sec), GPU_IDLE_CAUSE (sec). CPU_IDLE time and CPU_IDLE_CAUSE time is calculated in callbacks to *OpenCL* synchronization blocks. At the end of a synchronization block, *HPCToolkit* performs two operations: records CPU idle time and attributes it to the CCT of the synchronization call, and gets the list of completed kernels and attributes a portion of the CPU idle blame to each kernel in this list. GPU_IDLE_CAUSE is calculated using a Linux timer callback on each application thread: the callback will increment the GPU idle time for a CPU region by the timer interval if there are zero running kernels at the time the callback was triggered.

With the help of these blame-shifting metrics users can identify the GPU execution regions that cause CPU to sit idle (CPU_IDLE and CPU_IDLE_CAUSE) and, conversely, identify the CPU regions that cause the GPU to lay dormant (GPU_IDLE_CAUSE). With the former, the user can work on reducing the critical path of the application. With the latter, the application developer can offload some of the CPU computations to the GPUs so as to keep all compute resources busy at most times and thus utilize the compute bandwidth at disposition more fruitfully. Thus blame shifting helps reduce the overall execution time better than hotspot analysis. E.g. figure 2 shows a trace view for an application called *Amr-Wind*. This figure shows that GPU is idle for $\sim 97.67\%$ when the CPU thread executes clBuildProgram (this is the *OpenCL* routine that compiles and links *OpenCL* program executable from the program source or binary). We get the same results in profile view with the help of GPU_IDLE_CAUSE metric. Table. I shows the CPU functions with highest value for GPU_IDLE_CAUSE metric for the *Amr-Wind* application. All three functions in the table internally call clBuildProgram.

B. *Identifying potential inefficiencies*

GPU vendors release documents containing programming best practices for their GPU hardware. Users find it hard to find and apply the right optimizations in an application by referring to such documentation. Here, we describe how support in *HPCToolkit* detects that good practices described by the Intel GPU optimization guide [5] are not being followed.



```
void
Amr::FinalizeInit (Real             strt_time,
                   Real             stop_time)
{
    BL_PROFILE("Amr::FinalizeInit()");
    //
    // Compute dt and set time levels of all grid data.
    //
    amr_level[0]->computeInitialDt(finest_level,
                   sub_cycle,
                   n_cycle,
                   ref_ratio,
                   dt_level,
                   stop_time);
```

```
Scope                                                          KERNEL_PARAMS_NOT_ALIASED:Sum (I)
Experiment Aggregate Metrics                                                                  7
  <program root>                                                                              7
    529: main                                                                                 7
      127: amrex::Amr::init(double, double)                                                   7
        1120: [I] amrex::Amr::initialInit(double, double, amrex::BoxArray                     7
          1248: amrex::Amr::FinalizeInit(double, double)                                      7
            loop at AMReX_Amr.cpp: 1327                                                       4
              1328: PeleC::sum_integrated_quantities()                                        4
                loop at SumIQ.cpp: 28                                                         4
            1289: PeleC::computeInitialDt(int, int, amrex::Vector<int, st                     3
              loop at PeleC.cpp: 1052                                                         3
```

Fig. 3. Kernels in *PeleC* that can benefit from enabling code reordering

a) JIT compilation on single devices: A *DPC++* application binary will internally store the kernels in an intermediate format called SPIR-V. When it is time to offload the kernel to a target hardware device, the SPIR-V code is converted to assembly format according to the instruction set of the target hardware. This conversion is called Just-in-time (JIT) compilation. Note that JIT compilation takes up some overhead for each kernel when its being compiled for a new device. If the user has a single hardware device, then the cost of JIT compilation can be avoided by using ahead-of-time (AOT) compilation. *HPCToolkit* gets the device utilization count by intercepting clCreateContext and suggests when it is right to turn on AOT compilation with the help of the metric SINGLE_DEVICE_USE_AOT_COMPILATION.

b) Redundant memory transfers: Output computations of kernels are transferred to the host. These transfers are called device to host (D2H) transfers. It is possible that the output buffers are sent back to the accelerator device (after some pre-processing). These transfers are called host to device (H2D) transfers. One can avoid these D2H and H2D transfers altogether if the data massaging can be done on the device side. Memory transfers are expensive operations. The memory bandwidth is a bottleneck for GPU computations and avoiding redundant transfers will definitely improve the overall execution efficiency.

HPCToolkit intercepts all H2D and D2H calls to check if a buffer passed from GPU (via D2H) is sent as input again to GPU (via H2D) and brings such redundant transfers to the notice of the user with the help of the metric OUTPUT_OF_KERNEL_INPUT_TO_ANOTHER_KERNEL.

c) Code reordering disabled: Kernels usually take buffers as input arguments. Intel's *DPCPP* compiler assumes that aliasing is always present amongst the arguments and doesn't perform code reordering for instructions containing the kernel arguments. The user can let the compiler know explicitly when there is no pointer aliasing via pragmas, which will help get the benefits of instruction reordering optimization.

HPCToolkit tracks all buffer arguments used for an *OpenCL* kernel and uses an algorithm for detecting overlap of the memory regions belonging to the buffers during the kernel dispatch. *HPCToolkit* uses the KERNEL_PARAMS_NOT_ALIASED metric for kernels that don't have pointer aliasing to identify kernels where code-reordering of accesses using kernel argument pointers is legal.

Figure 3 shows *HPCToolkit* identifying kernels being called from amrex::Amr::FinalizeInit() that have non-overlapping pointer arguments (with the KERNEL_PARAMS_NOT_ALIASED metric). Enabling code reordering for kernels with non-overlapping arguments may significantly improve performance. A user can add intel::kernel_args_restrict pragma inside the kernel definition to notify the compiler that code reordering of accesses using argument pointers is allowed.

d) Redundant kernel JIT costs: In *DPC++*, each kernel is passed to devices for execution. The runtime abstraction of devices are queues. Each queue uses a context that can store JIT'ed kernels and input buffers. Multiple queues can use the same context internally. The advantage of executing kernels inside the same context is that the JIT'ing cost is paid only once for each kernel. If the same kernel is executed across multiple contexts, the JIT cost paid for each context.

At each kernel dispatch call, the kernel is passed to a queue. *HPCToolkit* maintains an internal data-structure that contains a list of all queue contexts onto which a kernel has been dispatched. At each kernel dispatch, two tasks are done: this list is updated if a new context is used, and a check is done to see if the kernel has been passed to multiple contexts. *OpenCL* kernels that are passed to multiple contexts are identified with the metric KERNEL_TO_MULTIPLE_QUEUES_MULTIPLE_CONTEXTS. The user can merge all the executions for that kernel to a single context to avoid the duplicate JIT cost.

e) Serialized kernel executions: Each hardware device that executes a kernel is represented as a queue object by *DPC++* runtime. Each queue object has properties that represent execution and device properties. These properties are set when the queue object is created. One such property called execution mode determines if the kernels will be executed in-order or out-of-order. In-order mode ensures that kernels are executed in the order that they are submitted. Out-of-order mode enables the runtime to execute multiple kernels on the device in parallel. *HPCToolkit* will check for queues that have in-order execution mode turned on by checking the queue properties passed to the function clCreateCommandQueue/clCreateCommandQueueWithProperties (these are *OpenCL* functions that create the queue objects). Such queues are identified with the metric INORDER_QUEUE. If its not necessary for kernels in a queue to execute in order, then the user can toggle the execution mode of the queues.

f) Unused accelerator devices: A node may have multiple accelerator devices. For workloads that have a good compute to memory ratio, it makes sense to leverage all available accelerator devices. This helps parallelize computations and distribute overall workload. The host CPU device can also be used to run some of the kernel computations. The workload splits between accelerators should be according to the capability of the devices (not always an equal split). *HPCToolkit* searches for all available devices inside *DPC++* runtime (using *OpenCL* functions clGetPlatformIDs and clGetDeviceIDs) and how many are being used (by monitoring device utilization count in clCreateContext). If all devices are not getting used, *HPCToolkit* brings this to notice of the user using the metric ALL_DEVICES_NOT_USED.

V. Analyzing Kernel Performance

Kernels are building blocks of a GPU application and thus it is important to analyze their execution efficiency. At present, instrumentation is the only available mechanism to observe performance inside kernels on Intel GPUs. This will change. Intel is developing hardware support for instruction-level performance measurement that will become available in a future generation of its GPUs.[1].

Instrumentation generally has a large overhead compared to other techniques such as sampling. The brute force method for instrumentation would be to insert a probe for each instruction. This is not advisable since overhead incurred by instrumentation is proportional to the number of instrumentation probes inserted and executed in the binary. Also, there are limits on register usage for storing instrumentation results. Storing instrumentation results in memory will increase measurement overhead.

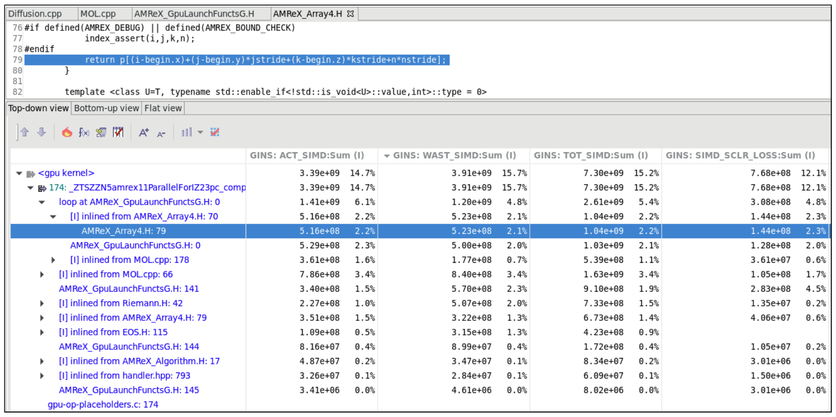
HPCToolkit uses Intel's *GT-Pin* library [11] to instrument the GPU kernels and gather performance information. *GT-Pin* has three callbacks: before compiling each kernel (GTPin_OnKernelBuild), before running each kernel (GTPin_OnKernelRun) and after each kernel execution (GTPin_OnKernelComplete). The *GT-Pin* instrumentation probes are inserted inside GTPin_OnKernelBuild callback time. GTPin_OnKernelRun ensures profiling is enabled and *HPCToolkit's* calling context data-structure is initialized for the kernel. Instrumentation results are collected and stored to the calling context in GTPin_OnKernelComplete.

HPCToolkit supports three types of kernel performance analysis: instruction execution counts, SIMD analysis, and latency analysis. We explain the strategies used to minimize instrumentation overhead for each of these measurements in their respective subsections. Running all three analysis modes simultaneously could alter instrumentation results. E.g. latency probe records bloated values because of SIMD probes. SIMD probes are not necessary inside latency analysis and therefore controlling probe insertion according to analysis needs improves the result accuracy and reduces instrumentation overhead.

A. Instruction execution counts

GTPin_OpcodeprofInstrument is the *GT-Pin* probe that gets the execution count for a GPU instruction. *HPCToolkit* inserts

[1]Intel has approved public release of this information.



```
#if defined(AMREX_DEBUG) || defined(AMREX_BOUND_CHECK)
            index_assert(i,j,k,n);
#endif
            return p[(i-begin.x)+(j-begin.y)*jstride+(k-begin.z)*kstride+n*nstride];
        }

        template <class U=T, typename std::enable_if<!std::is_void<U>::value,int>::type = 0>
```

	GINS: ACT_SIMD:Sum (I)		GINS: WAST_SIMD:Sum (I)		GINS: TOT_SIMD:Sum (I)		GINS: SIMD_SCLR_LOSS:Sum (I)	
<gpu kernel>	3.39e+09	14.7%	3.91e+09	15.7%	7.30e+09	15.2%	7.68e+08	12.1%
174: _ZTSZZN5amrex11ParallelForIZ23pc_comp	3.39e+09	14.7%	3.91e+09	15.7%	7.30e+09	15.2%	7.68e+08	12.1%
loop at AMReX_GpuLaunchFunctsG.H: 0	1.41e+09	6.1%	1.20e+09	4.8%	2.61e+09	5.4%	3.08e+08	4.8%
[I] inlined from AMReX_Array4.H: 70	5.16e+08	2.2%	5.23e+08	2.1%	1.04e+09	2.2%	1.44e+08	2.3%
AMReX_Array4.H: 79	5.16e+08	2.2%	5.23e+08	2.1%	1.04e+09	2.2%	1.44e+08	2.3%
AMReX_GpuLaunchFunctsG.H: 0	5.29e+08	2.3%	5.00e+08	2.0%	1.03e+09	2.1%	1.28e+08	2.0%
[I] inlined from MOL.cpp: 178	3.61e+08	1.6%	1.77e+08	0.7%	5.39e+08	1.1%	3.61e+07	0.6%
[I] inlined from MOL.cpp: 66	7.86e+08	3.4%	8.40e+08	3.4%	1.63e+09	3.4%	1.05e+08	1.7%
AMReX_GpuLaunchFunctsG.H: 141	3.40e+08	1.5%	5.70e+08	2.3%	9.10e+08	1.9%	2.83e+08	4.5%
[I] inlined from Riemann.H: 42	2.27e+08	1.0%	5.07e+08	2.0%	7.33e+08	1.5%	1.35e+07	0.2%
[I] inlined from AMReX_Array4.H: 79	3.51e+08	1.5%	3.22e+08	1.3%	6.73e+08	1.4%	4.06e+07	0.6%
[I] inlined from EOS.H: 115	1.09e+08	0.5%	3.15e+08	1.3%	4.23e+08	0.9%		
AMReX_GpuLaunchFunctsG.H: 144	8.16e+07	0.4%	8.99e+07	0.4%	1.72e+08	0.4%	1.05e+07	0.2%
[I] inlined from AMReX_Algorithm.H: 17	4.87e+07	0.2%	3.47e+07	0.1%	8.34e+07	0.2%	3.01e+06	0.0%
[I] inlined from handler.hpp: 793	3.26e+07	0.1%	2.84e+07	0.1%	6.09e+07	0.1%	1.50e+06	0.0%
AMReX_GpuLaunchFunctsG.H: 145	3.41e+06	0.0%	4.61e+06	0.0%	8.02e+06	0.0%	3.01e+06	0.0%
gpu-op-placeholders.c: 174								

Fig. 4. SIMD analysis inside *HPCToolkit*

this probe once for each basic-block, since all instructions within a basic-block will have the same execution count. EXC_CNT is the metric inside *HPCToolkit* that records the GPU instruction execution counts and aggregates it to the source code.

B. SIMD analysis

SIMD instructions operate on multiple data elements simultaneously. Intel's GPUs use SIMD instructions. With the help of SIMD analysis we want to understand how well each kernel utilizes SIMD lanes and identify the code regions in each kernel that incur significant SIMD waste.

1) Calculating SIMD waste: For measuring SIMD waste, *HPCToolkit* need two things: SIMD utilization and total SIMD lanes available. SIMD waste is the difference between total lanes and utilized lanes.

For calculating SIMD utilization for a kernel, we leveraged the code for SIMDProf [12]—an Intel *GT-Pin* tool. This tool calculates the SIMD values internally using instrumentation and instruction-specific variables such as instruction mask, predication value, etc. *GT-Pin* instrumentation probes for getting SIMD utilization values are GTPin_SimdProfInstrument. SIMDProf gives an approach to reduce the probes needed for SIMD analysis. Calculating SIMD utilization is dependent on certain static and dynamic instruction properties, such as execution mask, predication, SIMD channels available for the current instruction, etc. If we were to identify instruction groups such that all instructions in a group have same static and dynamic properties, we can add a single instrumentation probe for the entire group. With this single probe *HPCToolkit* can get the SIMD utilization for all instructions in the group. This logical grouping of instructions help reduce the total number of instrumentation probes for SIMD analysis.

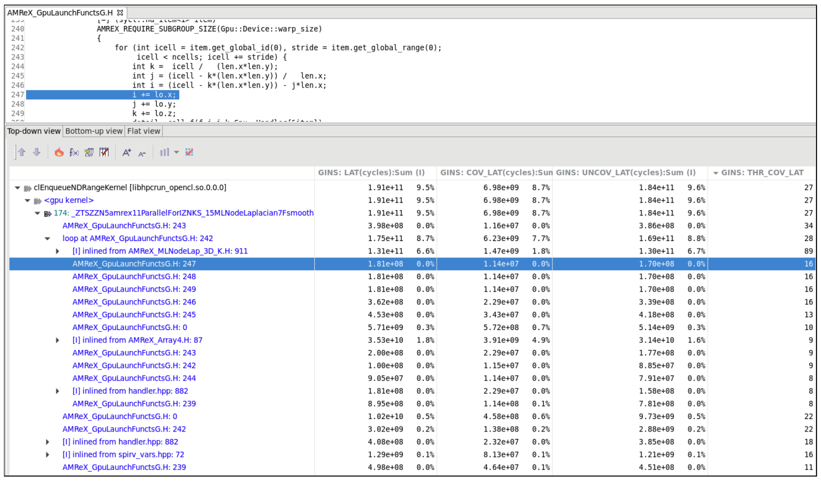
The second step is to calculate the total lanes available. We can calculate this by multiplying the instruction frequency with total SIMD lanes available for each instruction. Instruction frequency can be calculated with the help of instrumentation; total SIMD lanes can be queried via *GT-Pin* (GTPin_KernelGetSIMD). With all the necessary inputs in hand for our computation, *HPCToolkit* can calculate the SIMD waste for the GPU kernels.

Figure 4 shows a run for SIMD analysis for *PeleC* application inside *HPCToolkit*. *HPCToolkit* computes four metrics as part of SIMD analysis: active SIMD lanes (ACT_SIMD), wasted SIMD lanes (WAST_SIMD), total SIMD lanes (TOT_SIMD) and SIMD loss due to scalar instructions (SIMD_SCLR_LOSS). Metrics WAST_SIMD and SIMD_SCLR_LOSS are important with respect to optimization: with these the user can sort the results and find the regions incurring most SIMD waste. Once those regions are brought in view, the user can view the kernel source code and identify the reason for SIMD waste. With this knowledge, the user can apply the corresponding code transformation(s) to improve SIMD utilization. The highlighted row is one of the highest contributor to SIMD waste for pc_compute_hyp_mol_flux kernel. The source pane shows that the corresponding line is a return of an array value and as discussed before, array index calculations involve address arithmetic.

C. Latency analysis

Hotspot analysis gives us the execution time of the kernel. For expensive kernels, the user would like to know the fine-level execution details: why does the kernel take so long to execute, which are the most expensive lines in the kernel, etc. In this section, we give these execution details about the kernel. To be specific, *HPCToolkit* performs latency attribution for the instructions within a kernel, provides the efficiency of latency hiding inside the GPU and identifies the instructions that are culprits for large latency within the kernel.

For latency analysis, *HPCToolkit* inserts probes on instruction groups to reduce the instrumentation probe count. Instructions in a kernel are grouped into basic blocks. We can add a single instrumentation probe for each basic block to get both the execution frequency and latency for the basic block. Once we get the latency incurred for the entire basic block, we calculate instruction-level latency using statistical analysis as described in [13].


AMReX_GpuLaunchFunctsG.H

```
                AMREX_REQUIRE_SUBGROUP_SIZE(Gpu::Device::warp_size)
                {
                    for (int icell = item.get_global_id(0), stride = item.get_global_range(0);
                         icell < ncells; icell += stride) {
                        int k =  icell /   (len.x*len.y);
                        int j = (icell - k*(len.x*len.y)) /   len.x;
                        int i = (icell - k*(len.x*len.y)) - j*len.x;
                        i += lo.x;
                        j += lo.y;
                        k += lo.z;
```

	GINS: LAT(cycles):Sum (I)		GINS: COV_LAT(cycles):Sun		GINS: UNCOV_LAT(cycles):Sum (I)		GINS: THR_COV_LAT
clEnqueueNDRangeKernel [libhpcrun_opencl.so.0.0.0]	1.91e+11	9.5%	6.98e+09	8.7%	1.84e+11	9.6%	27
<gpu kernel>	1.91e+11	9.5%	6.98e+09	8.7%	1.84e+11	9.6%	27
174: _ZTSZZN5amrex11ParallelForIZNKS_15MLNodeLaplacian7Fsmooth	1.91e+11	9.5%	6.98e+09	8.7%	1.84e+11	9.6%	27
AMReX_GpuLaunchFunctsG.H: 243	3.98e+08	0.0%	1.16e+07	0.0%	3.86e+08	0.0%	34
loop at AMReX_GpuLaunchFunctsG.H: 242	1.75e+11	8.7%	6.23e+09	7.7%	1.69e+11	8.8%	28
[I] inlined from AMReX_MLNodeLap_3D_K.H: 911	1.31e+11	6.6%	1.47e+09	1.8%	1.30e+11	6.7%	89
AMReX_GpuLaunchFunctsG.H: 247	1.81e+08	0.0%	1.14e+07	0.0%	1.70e+08	0.0%	16
AMReX_GpuLaunchFunctsG.H: 248	1.81e+08	0.0%	1.14e+07	0.0%	1.70e+08	0.0%	16
AMReX_GpuLaunchFunctsG.H: 249	1.81e+08	0.0%	1.14e+07	0.0%	1.70e+08	0.0%	16
AMReX_GpuLaunchFunctsG.H: 246	3.62e+08	0.0%	2.29e+07	0.0%	3.39e+08	0.0%	16
AMReX_GpuLaunchFunctsG.H: 245	4.53e+08	0.0%	3.43e+07	0.0%	4.18e+08	0.0%	13
AMReX_GpuLaunchFunctsG.H: 0	5.71e+09	0.3%	5.72e+08	0.7%	5.14e+09	0.3%	10
[I] inlined from AMReX_Array4.H: 87	3.53e+10	1.8%	3.91e+09	4.9%	3.14e+10	1.6%	9
AMReX_GpuLaunchFunctsG.H: 243	2.00e+08	0.0%	2.29e+07	0.0%	1.77e+08	0.0%	9
AMReX_GpuLaunchFunctsG.H: 242	1.00e+08	0.0%	1.15e+07	0.0%	8.85e+07	0.0%	9
AMReX_GpuLaunchFunctsG.H: 244	9.05e+07	0.0%	1.14e+07	0.0%	7.91e+07	0.0%	8
[I] inlined from handler.hpp: 882	1.81e+08	0.0%	2.29e+07	0.0%	1.58e+08	0.0%	8
AMReX_GpuLaunchFunctsG.H: 239	8.95e+08	0.0%	1.14e+08	0.1%	7.81e+08	0.0%	8
AMReX_GpuLaunchFunctsG.H: 0	1.02e+10	0.5%	4.58e+08	0.6%	9.73e+09	0.5%	22
AMReX_GpuLaunchFunctsG.H: 242	3.02e+09	0.2%	1.38e+08	0.2%	2.88e+09	0.2%	22
[I] inlined from handler.hpp: 882	4.08e+08	0.0%	2.32e+07	0.0%	3.85e+08	0.0%	18
[I] inlined from spirv_vars.hpp: 72	1.29e+09	0.1%	8.13e+07	0.1%	1.21e+09	0.1%	16
AMReX_GpuLaunchFunctsG.H: 239	4.98e+08	0.0%	4.64e+07	0.1%	4.51e+08	0.0%	11

Fig. 5. Latency analysis inside *HPCToolkit*

1) Latency attribution: GT-Pin instrumentation probes for getting latency values are GTPin_LatencyInstrumentPre and GTPin_LatencyInstrumentPost_Mem. These probes are inserted for each basic-block and we then employ statistical analysis to get instruction-level latency. Combining instruction latency with line-map information retrieved from the GPU binary helps us to portray the latency for the source lines within the kernel. With this functionality integrated inside *HPCToolkit*, the user can view the most expensive source lines for the GPU kernels.

Figure 5 shows latency analysis done for *Amr-Wind* [14] inside *HPCToolkit*. Kernel Node_Laplacian is one of the most expensive kernels inside *Amr-Wind*. There are 4 metrics involved in latency analysis: latency cycles (LAT), covered latency (COV_LAT), uncovered latency (UNCOV_LAT) and threads needed to cover latency (THR_COV_LAT). Latency cycles helps us view the latency accumulated at source line level, with this the user gets finer execution perspective.

2) Latency hiding: Latency hiding is a technique used inside CPUs and GPUs to hide idleness and keep the hardware busy as much as possible. Latency hiding is done using multi-threading inside Intel GPUs (refer section II for more detailed explanation). The formula to calculate threads needed to hide latency inside a GPU execution unit (EU) is: $1 + (U/C)$, (U : uncovered latency, C : covered latency). We calculate C and U with help of latency values calculated for kernel instructions. Latency is sum of covered and uncovered latency. We can approximate C as 1 cycle for each instruction. U is the difference between latency and C. Once we have these values, we can plug it into the above formula to get the number of threads needed in the EU to fully hide latency. If this value is $<= 7$ (the number of hardware threads present in Intel Gen9 GPUs [5]), then all of the uncovered latency will be hidden with the help of multi-threading. If the number of threads needed is higher than seven, the user needs to explore ways to optimize his kernels and reduce uncovered latency. This can be done with the help of latency attribution and latency blaming.

```
for (all edges in the def-use graph for a kernel) {
    i = from_vertex;
    j = to_vertex;

    $\text{latency\_blame(i)} = \dfrac{\text{reciprocal\_of\_path\_length(i) * execution\_frequency(i)}}{\sum_{(k \in \text{parents(j)})} \text{(reciprocal\_of\_path\_length(k) * execution\_frequency(k))}} \text{ * latency(j)}$
}
```

Fig. 6. Latency blaming with help of def-use relation between instructions

Columns 2, 3 and 4 in Figure 5 represent covered latency, uncovered latency and threads needed to hide latency.

3) Latency blaming: Users can use latency attribution (LAT column from figure 5) to see the most expensive lines in their kernel. But this metric doesn't always paint the accurate picture of kernel execution. An instruction (or source line) may be taking a lot of execution cycles, but that could be because it is waiting for the results from a preceding instruction (line). In that case, the real culprit of latency is the predecessor.

For latency blaming, we need latency values for the kernel (calculated in latency attribution), execution frequency (using instrumentation) and the def-use relationship between kernel instructions. We get the def-use graph for kernels with the help of the Dyninst library [15] and IGA [16].

Figure 6 describes the formula we use for latency blaming. This formula is a modification of the stall blame analysis done in GPA [17]. The latency blame for the def instruction is calculated with help of latency values of the use instructions, execution frequency of both def and use, and the distance between the def and use instruction. Note that the greater the path length between source (def) and destination (use) vertices, lesser the blame that gets accumulated to the source vertex.

With the latency blame view inside *HPCToolkit*, the user can take action to optimize the source lines that are the actual culprits of high latency in the kernel.

D. Attributing measurements to program source

HPCToolkit's hpcstruct program is used to analyze Intel GPU binaries. Hpcstruct uses Intel's IGA library [16] to parse the binaries and read the machine instructions. By identifying branch instructions and branch targets, it reconstructs the CFG of the binary. This CFG is then passed internally onto dyninst which recovers the loop-nesting structure for the binary. Then hpcstruct identifies the nesting of the machine instructions inside the loops that are present in the code. It also looks at the attribution of each machine instruction back to the program source, for which it is dependent on the line-map information made available by the compiler.

GPU kernels, just like CPU functions, could invoke other kernels, lambdas, inline functions, etc. If GPU compilers do not adequately capture these calls for optimized binaries, profiling tools cannot attribute the measured metrics to the correct source code locations. While evaluating the kernel analysis results run on Intel GPUs with *DPC++* compilers (developed by Intel), we observed that the call chains in the GPU context was not correctly captured by the compiler. This could result in *HPCToolkit* displaying misleading profiling results in some cases. At the time of writing this paper, the Intel compiler team has been notified of this issue.

VI. Case Studies

In the previous sections, we described capabilities *HPCToolkit's* support for measurement and analysis of GPU-accelerated applications executing on Intel GPUs atop *OpenCL*. In this section, we use two applications to demonstrate *HPCToolkit's* utility for measurement and analysis of sophisticated applications.

A. PeleC

PeleC [18] [19] is a combustion application built by LBNL, NREL, and ANL, tailored for supercomputers and exascale machines. It uses Direct Numerical Simulations (DNS) of turbulence-chemistry interactions in real-world conditions, models system structure with Embedded Boundary (EB) capability and uses AMR (Adaptive Mesh Refinement) which is a system for mesh refinement based on the AMReX framework. We studied *PeleC's* Taylor-Green Vortex (TG) problem written in *DPC++*.

With *HPCToolkit's* latency analysis, we observed that a for-loop executed as part of kernel pc_compute_diffusion_flux had a lot of uncovered latency. This loop had three variables: i, j, k. These variables were initialized in order $k \rightarrow j \rightarrow i$ but then were immediately consumed in the order $i \rightarrow j \rightarrow k$ by subsequent lines. This code was part of the AMReX library. Upon closer inspection, we observed the same usage pattern for multiple AMReX function routines. If the initialization order was maintained during consumption, we could observe more latency hiding (Intel compiler could be doing this reordering in the binary, but it doesn't hurt to manually do such optimizations). SIMD analysis run on *PeleC* shows most of the SIMD waste occurring at device function invocations, loop conditionals, address arithmetic, etc. 25.5% of the SIMD waste is due to scalar instructions. Unsurprisingly, the top two expensive kernels inside the application, pc_compute_hyp_mol_flux and pc_compute_diffusion_mol_flux, contribute to $\sim 67\%$ of the SIMD waste.

Blame-shifting analysis for *PeleC* shows the top candidate kernels for optimization (pc_compute_hyp_mol_flux, pc_compute_diffusion_mol_flux, Diffusion, getMOLSrcTerm, copy_array and setV).

Of the two program inefficiencies identified for *PeleC*, first was to enable ahead-of-time (AOT) compilation or offload kernels to CPU (host) device. There were two compute devices available in the remote server that ran the experiment: an Intel GPU (Iris(R) Pro Graphics P580) and Intel CPU (Xeon(R) CPU E3-1585 v5 @ 3.50GHz). The second suggestion given for optimization was to enable code reordering in kernels that don't have pointer aliasing. Our analysis showed that seven kernel executions could benefit from this optimization (Figure 3). All seven kernels are called from loop regions. Thus enabling this optimization can help improve the kernel performance.

B. Amr-Wind

ExaWind [20] is an open-source collection of physics codes and libraries for multi-fidelity wind turbine and wind power plant simulation written to harness exascale computational power. We studied the *Amr-Wind* [14] application inside the ExaWind suite, which is a massively parallel, block-structured adaptive-mesh, and in-compressible flow solver for wind turbine and wind farm simulations. The solver is built on top of the AMReX library. For our case study, we worked with the *DPC++* version of the application.

For latency analysis, we observed that a for-loop in kernel amrexParallelForNodeLaplacianFsmooth lost many cycles in uncovered latency. This loop had the same inverted initialization-usage pattern that we observed for other AMRex functions inside *PeleC*. Correcting this pattern across the AMRex library could potentially lower execution time for many applications importing this library in their code-base. SIMD analysis shows that SIMD loss is occurring at different regions: at address arithmetic instructions, conditional checks, etc. Most of the lines showing SIMD waste consisted of address arithmetic logic. 16% of the SIMD loss is contributed by scalar instructions (instructions that are using a single SIMD lane).

When blame-shifting was run on *Amr-Wind*, we observed that it is poorly utilizing the GPU compute power. The GPU was idle for $\sim 97.67\%$. Taking a deeper look, we see that $\sim$ 510 seconds are spent in clBuildProgram, which is an *OpenCL* function responsible for building the program binaries. When the slowdown issue was raised with a developer from the *Amr-Wind* team, they accepted this issue and suggested that adding the build option -fsycl-device-code-split=per_kernel reduces the JIT compilation time during execution. *HPCToolkit* requires -g flag to be passed in the application to extract debugging information from the binaries. However, adding -fsycl-device-code-split=per_kernel along with -g flag caused

TABLE II
ANALYSIS OVERHEAD

Application	Vanilla run sec	SIMD analysis sec(overhead)	Latency analysis sec(overhead)	Blame shifting sec(overhead)	Program inefficiencies sec(overhead)
PeleC	8.617	14.497(1.68x)	13.695(1.59x)	10.418(1.21x)	10.376(1.2x)
Amr-Wind	519.215	1295.975(2.5x)	1243.127(2.39x)	911.591(1.75x)	880.857(1.7x)

the application build to fail. Another alternative suggested to mitigate this slowdown is using AOT compilation. This direction is yet to be explored.

Program inefficiencies identified for *Amr-Wind* are similar to the ones for *PeleC*. First was to enable AOT compilation or offload kernels to CPU (host) device. If the user offloads more of the computational load to kernels, it would make sense to use either AOT compilation or both compute devices to gain speedup. The second suggestion given for optimization was to enable code reordering in kernels that don't have pointer aliasing. Our analysis showed that ~ 9000 kernel executions could benefit from this optimization. Many of these kernels are invoked from loops. Since a relatively small portion of our execution of an *Amr-Wind* benchmark is spent executing GPU kernels, the benefits of these optimizations will be visible only when kernel execution takes up more of the computation load.

C. Measurement Overhead

HPCToolkit uses binary instrumentation for SIMD and latency analysis because Intel GPUs do not support hardware-based instruction measurement at present. The instrumentation overhead is reduced with the help of sparse probes and statistical analysis.

Table II lists the execution results of our analysis suite with two *DPC++* applications: *PeleC* [18] and *Amr-Wind* [14].

The overhead of our analysis ranges from $1.2\times$–$2.5\times$. The highest overhead is observed for SIMD analysis, followed by latency analysis. Our tool's overhead is comparable to the previous work's $1.01\times$–$2.5\times$ overheads that also uses GT-Pin for instrumentation [13]. The overhead incurred by our analysis lies within this range which implies that our overhead is acceptable. Intel VTune's [21] latency and SIMD analysis on *Amr-Wind* had $2.1\times$ and $2.14\times$ overhead respectively. VTune's run did kernel profiling, not application profiling. So the overhead is not entirely comparable. NVIDIA's CUPTI instrumentation on the other hand has shown overheads up to 296x for GPU applications [22].

VII. Related Work

VTune [21] is a commercial profiling tool provided by Intel. It supports process and system-level profiling. Process-level profiling includes fine-grained measurements using GTPin such as SIMD usage, latency values, etc. It has a profile and timeline view and provides source code correlation of the collected profiles. Inside the source view, the GPU instruction count, SIMD utilization, latency cycles can be viewed. Some additional features in *VTune* are: analyzing cache performance, FPGA profiling and assembly code correlation to metrics. Some disadvantages of *VTune* are it does not merge CPU and GPU calling-context view of metrics and, it gives profiling support only for Intel GPUs.

Nsight Compute [23] is NVIDIA's commercial profiling tool for NVIDIA GPUs. Some of its features are roofline model analysis, memory usage representation, profile view with source code and assembly correlation, and PC sampling. Unlike NVIDIA's tools, *HPCToolkit* associates instruction-level performance metrics with detailed contexts including inlined code, loops, device function calls, and source lines. In contrast, NVIDIA's tools only relate instruction-level costs to kernel source lines. Such a view is almost useless for highly inlined code because it is not apparent how source lines from deeply nested chains of inlined functions came to be present in a kernel.

ROC profiler library [24] is AMD's GPU profiling API. It provides profiling support using hardware counters, derived metrics and application tracing with which users can view kernel execution, async memory transfers and barrier details. *Rocprof* is a command-line tool that uses this API internally.

TAU [25] is a profiling tool for *OpenCL* and *CUDA* developed by University of Oregon, Los Alamos National Laboratory, and Research Centre Juelich. It consists of a profile view and trace view as well as support for instrumentation analysis for CPUs. It provides a flat-profile for GPUs at kernel level and does not give a calling-context view.

VIII. Conclusions and Future Work

This paper describes performance measurement and analysis support in *HPCToolkit* for *DPC++/OpenCL* applications running on Intel GPUs.

HPCToolkit leverages blame-shifting to identify the non-overlapped CPU-GPU execution. Moreover, *HPCToolkit* checks various inefficiencies in GPU kernels and provides corresponding optimization suggestions. *HPCToolkit* also provides fine-grained kernel insights such as instruction execution counts, SIMD analysis, and latency hiding analysis.

At run-time, *HPCToolkit* currently processes measurements collected using *GT-Pin* instrumentation and attributes them at both the basic block and instruction level. The overhead incurred due to instruction-level attribution is substantial. For *Amr-Wind*, it was over 6x the cost associated with basic block level attribution. The runtime overhead of instruction-level attribution can be avoided by deferring it to post-mortem analysis. During post-mortem analysis, the overhead of instruction-level attribution would be less since it would be incurred once for each kernel calling context rather than once for each kernel invocation.

At the time of this writing, hardware support for instruction-level measurement of GPU performance is only available on NVIDIA GPUs. Intel is developing hardware support for instruction-level performance measurement that will become available in a future generation of its GPUs. The addition of hardware support for instruction-level measurement should provide complementary metrics to instrumentation-based metrics. Currently, *HPCToolkit* attributes instruction metrics with incomplete inline information. Once Intel compilers start capturing inline calls made inside GPU binaries, users would be able to see a clearer picture of the kernel execution and understand avenues to add optimizations to the kernels.

We will enhance *HPCToolkit* to provide more intuitive guidance for inefficiencies. First, we plan to provide an estimated performance gain associated with each inefficiency, e.g., the time savings associated with avoiding redundant JIT compilation. Also, the program inefficiencies are currently visible as metrics in columnar format. We want to give these suggestions to the user in a GUI-friendly and intuitive fashion by borrowing ideas from existing applications. Last, we would like to extend our tool with instruction-level analysis to assess important code characteristics such as GPU register pressure.

Eventually, Intel plans to fully transition from *OpenCL* to its emerging *Level Zero* runtime but at present supports both. The interface for performance monitoring in *Level Zero* is similar to that for *OpenCL*, so we expect to simply transition to new interfaces for monitoring offloading using *Level Zero* and leveraging the core monitoring and analysis functionality described in this work for programs running atop *Level Zero*.

Acknowledgments

This research was supported in part by the Exascale Computing Project (17-SC-20-SC)—a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, Argonne National Laboratory (Subcontract 9F-60073), and Advanced Micro Devices.

References

[1] Argonne Leadership Computing Facility, "Aurora." https://www.alcf.anl.gov/aurora.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[3] K. Zhou, M. W. Krentel, and J. Mellor-Crummey, "Tools for top-down performance analysis of gpu-accelerated applications," in *Proceedings of the 34th ACM International Conference on Supercomputing*, pp. 1–12, 2020.

[4] D. Blythe, "Intel's Ponte Vecchio GPU architecture." https://hc33.hotchips.org/assets/program/conference/day2/hc2021_pvc_final.pdf, 2021.

[5] Intel Corporation, "oneAPI GPU optimization guide." https://software.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide/top.html, 2020.

[6] D. Callahan, S. Carr, and K. Kennedy, "Improving register allocation for subscripted variables," *SIGPLAN Not.*, vol. 39, p. 328–342, Apr. 2004.

[7] M. Chabbi, K. Murthy, M. Fagan, and J. Mellor-Crummey, "Effective sampling-driven performance tools for GPU-accelerated supercomputers," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2013.

[8] Khronos Group, "Opencl execution model." https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf, 2012.

[9] Khronos SYCLWorking Group, "SYCL 2020 Specification (revision 3)." https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

[10] Jim Valerio, "GPU Compute Engine: Theory of operation. distribution restricted by NDA."

[11] M. Kambadur, S. Hong, J. Cabral, H. Patil, C.-K. Luk, S. Sajid, and M. A. Kim, "Fast computational GPU design with GT-Pin," in *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, IISWC '15, (USA), p. 76–86, IEEE Computer Society, 2015.

[12] Intel Corporation, "Simdprof sample tool." https://software.intel.com/sites/landingpage/gtpin/_s_i_m_d_p_r_o_f__t_o_o_l.html, 2020.

[13] A. V. Gorshkov, M. Berezalsky, J. Fedorova, K. Levit-Gurevich, and N. Itzhaki, "GPU instruction hotspots detection based on binary instrumentation approach," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1213–1224, 2019.

[14] Lawrence Berkeley National Laboratory, National Renewable Energy Laboratory, and Sandia National Laboratories, "AMR-Wind: a massively parallel, block-structured adaptive-mesh, incompressible flow solver for wind turbine and wind farm simulations." https://github.com/Exawind/amr-wind, 2018.

[15] Computer Sciences Department, University of Wisconsin–Madison, "Dyninst DataflowAPI programmer's guide." https://dyninst.org/sites/default/files/manuals/dyninst/dataflowAPI.pdf.

[16] Intel Corporation, "IGA: Intel Graphic Assembler." https://github.com/intel/intel-graphics-compiler/blob/master/visa/iga/IGALibrary/api/kv.h.

[17] K. Zhou, X. Meng, R. Sai, and J. Mellor-Crummey, "Gpa: A gpu performance advisor based on instruction sampling," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 115–125, IEEE, 2021.

[18] National Renewable Energy Lab. (NREL) and Lawrence Berkeley National Lab, "Pelec: an adaptive-mesh compressible hydrodynamics code for reacting flows." https://github.com/AMReX-Combustion/PeleC, 2018.

[19] S. Whitman, J. Brasseur, and P. Hamlington, "Simulation of bluff-body stabilized flames with PeleC, an Exascale combustion code." https://bluewaters.ncsa.illinois.edu/liferay-content/document-library/19symposium-slides/whitman_flames.pdf, 2018.

[20] Wind Energy Technologies Office, Office of Energy Efficiency & Renewable Energy, "ExaWind supercharges wind power plant simulations on land and at sea." https://www.energy.gov/eere/wind/articles/exawind-supercharges-wind-power-plant-simulations-land-and-sea, 2020.

[21] A. Marowka, "On performance analysis of a multithreaded application parallelized by different programming models using Intel VTune," in *International Conference on Parallel Computing Technologies*, pp. 317–331, Springer, 2011.

[22] K. Zhou, X. Meng, R. Sai, D. Grubisic, and J. M. Mellor-Crummey, "An automated tool for analysis and tuning of gpu-accelerated code in hpc applications," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[23] NVIDIA Corporation, "NVIDIA Nsight Compute: an interactive kernel profiler for CUDA applications." https://developer.nvidia.com/nsight-compute.

[24] Advanced Micro Devices, Inc, "AMD ROCm ROCProfiler." https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCm-Tools.html.

[25] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.