Comprehensive Performance Monitoring for GPU Cluster Systems

Karl Fürlinger
Ludwig-Maximilians-Universität (LMU) Munich
Computer Science Department, MNM Team
Oettingenstr. 67, 80538 Munich, Germany
Email: fuerling@nm.ifi.lmu.de

Nicholas J. Wright and David Skinner
NERSC
Lawrence Berkeley National Laboratory
Berkeley, California 94720, USA
Email: {njwright, deskinner}@lbl.gov

Abstract—**Accelerating applications with GPUs has recently garnered a lot of interest from the scientific computing community. While tools for optimizing individual kernels are readily available, there is a lack of support for the specific needs of the HPC area. Most importantly, integration with existing parallel programming models (MPI and threading) and scalability to the full size of the machine are required. To address these issues we present our work on monitoring and performance evaluation of the CUDA runtime environment in the context of our scalable and efficient profiling tool IPM. We derive metrics for GPU utilization and identify missed opportunities for GPU-CPU overlap. We evaluate the monitoring accuracy and overheads of our approach and apply it to a full scientific application.**

***Keywords*-performance monitoring and analysis; GPGPU computing; NVIDIA CUDA; GPU clusters; efficient scalable monitoring**

I. Introduction

Heterogeneous and accelerator-based systems are gaining a lot of momentum in the high performance computing area. The addition of ECC protection for device memory and a significant boost in double precision floating point performance has rendered NVIDIA graphics cards an attractive building block for supercomputers. As of December 2010 the world's fastest computer according to the Top 500 list (the Chinese *Tianhe-1A* system) as well as numerous smaller clusters are equipped with GPUs. Future massively parallel leadership-class systems in the multi-Petaflop range are likely to come with accelerators as well [1].

However, the optimal choice of programming model for heterogeneous accelerator-based systems is quite far from being determined. A developer has several ways to offload all or a portion of their application onto the accelerator. The options range from leveraging accelerated libraries (like CUBLAS, CUFFT, CULA [2], MAGMA [3]), decorating kernels with accelerator pragmas and relying on an accelerator-aware compilation tool-chain (e.g., PGI [4], [5], HMPP [6]), to the manual implementations of kernels specifically tuned for the accelerator, in either CUDA or OpenCL. All of these scenarios require the developer to have solid performance feedback in order to assess tuning opportunities and direct optimization strategies.

In this paper we describe our work on providing a comprehensive performance monitoring solution for applications executed on GPU clusters in the context of our performance monitoring and analysis tool IPM [7], [8]. We focus on CUDA as we see this framework currently having the most momentum. Our goal is to enable developers who design, adapt, or port their parallel codes to GPU clusters to get feedback from execution at the full scale of machines. Notably, we argue that optimization of individual kernels for a single workstation will not be sufficient for the effective use of accelerators in an HPC setting. We believe that monitoring at the full scale of machines is needed and that the interplay of various levels of parallelism (MPI, OpenMP) and other performance-influencing factors needs to be accounted for as well.

Here is a list of some of the issues that a programmer has to address to create a well-performing MPI+CUDA parallel application for GPU clusters

1) Each kernel needs to be optimized for the GPU individually.
2) The scheduling and interaction of multiple kernels has to be analyzed.
3) The load balancing across the MPI processes needs to be checked. As the number of MPI processes increases, the fraction of time spent in computation typically decreases and the MPI communication time increases. A smaller dataset for GPU offloading can also mean that the time to transfer data via the PCIe bus in relation to the GPU compute time changes, and offloading might become less beneficial.
4) While in the sequential case, GPU activity can only be overlapped with other CPU compute activities or file-I/O, in a parallel setting MPI communication with other processes is an additional option.
5) GPU cluster configurations vary widely. In some installations, a single GPU is paired with one or several multi-core CPUs and multiple MPI tasks might have to share the single GPU. In other scenarios a GPU can be exclusive to an MPI task. In the shared GPU case, the kernel performance might be dramatically different in the production MPI case compared to an isolated

workstation setting.

6) There are more factors beyond the immediate control of a developer on large-scale machines. The overall system load, file-system activity, background daemons and stray processes are impossible to predict but influence the application execution. Therefore, the application has to be measured on the actual machine to explain performance and scaling behavior and factor out various influencing factors.

Some of these issues, such as (1) and (2) can be addressed with existing tools such as the CUDA profiler or Parallel Nsight. Others such as (3)–(6) can only be analyzed when the parallel application as a whole is taken into account, which is the focus of our work.

The main contributions of this article are as follows:

- We describe an effective monitoring solution for CUDA, entirely implemented as a thin layer interposed between the application and the CUDA runtime. No source code changes, recompilation, or even re-linking of the application is required.
- We show how basic host-side timing can be extended to uncover kernel execution time on the GPU and we define a metric of implicit host blocking time to identify opportunities for overlapped execution.
- We provide a monitoring layer for accelerated numerical libraries (BLAS and FFT), to allow developers that try to leverage accelerators through re-linking with optimized libraries to obtain performance information.

The rest of this paper is organized in the following way: To set the stage we give a short overview of our existing IPM (integrated performance monitoring) tool set in Section II. Then, Section III describes our monitoring methodology for CUDA in detail. Our approach is based on intercepting the runtime library calls and we show through a series of examples how basic host (CPU) side timing can be augmented to derive valuable metrics about GPU utilization and GPU-CPU interactions. In Section IV we evaluate our approach with respect to monitoring fidelity and overheads with several applications, ranging from small benchmarks to actual scientific applications. We review the related work in Section V and conclude in Section VI.

II. An Overview of IPM

IPM is a highly scalable workload characterization and performance monitoring tool. Originally focused on MPI, it has recently been extended to cover a number of other domains such as OpenMP and file-I/O [9]. IPM acts as a thin measurement layer between the application and the operating system and runtime. Its goal is to obtain the complete runtime *event inventory* and to derive high-level application characteristics (such as the communication percentage or the parallel coverage) from it. IPM's implementation strives to minimize monitoring overhead and application perturbation. A recent study has shown that the application perturbation is typically less than 0.5% of overall execution time, which is often less than the natural system variability due to system noise. [9] In contrast to a traditional *ad-hoc* performance analysis setting, this enables a scenario where the monitoring can be activated for each job executed on a high performance cluster. A project investigating this approach is currently under way at the NERSC computing center.

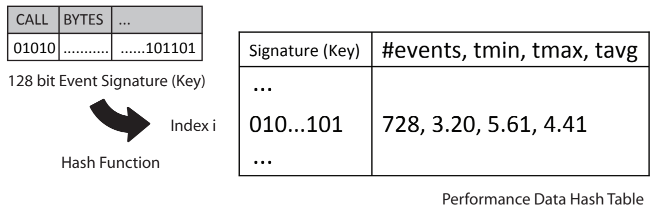


Figure 1: Event attributes and the hash table used by IPM.

During application execution, IPM's main mechanism for storing data is a central performance data hash table. The structure of the performance data hash table is outlined in Fig. 1. The hash key (also called the event signature) is derived from the type of monitored event (e.g., `MPI_Send` or `fopen`) as well as a number of other attributes such as the number of bytes transmitted or read. For each hash table entry IPM stores the number of calls made and the average duration, as well as the minimum and maximum for each call.

The output of IPM comes in two forms. Immediately after program termination a profiling banner report is written to stdout. This report summarizes the most important aspects of the execution. IPM also writes a more detailed profiling log in XML format which includes the full details of the hash table. The XML file can then be used by the IPM parser (`ipm_parse`) to produce a number of different output formats. The parser can re-produce the banner, it can generate an HTML based webpage (which is well-suited for permanent storage of the profiling report), and it can convert the IPM profile into the CUBE format [10]. The last option is a recent addition and is particularly well suited for the interactive exploration of performance data using the CUBE GUI. CUBE is part of the Scalasca tool set [10] and also available for download as stand-alone package[11]. In previous work [9] we have also demonstrated IPM's scalability up to the full size of current supercomputers (several tens of thousands of cores).

III. CUDA Monitoring with IPM

The CUDA programming system consists of a compiler (nvcc), a runtime system, and a device driver managing the interaction with the GPU. nvcc translates CUDA kernel code into PTX assembly which is just-in-time compiled for the GPU by the driver. The runtime system manages the memory transfers (which can be either synchronous

```
cudaError_t cudaCall(arg1,...) {
  cudaError_t ret;
  double begin, end, duration;

  begin = get_time();
  ret = real_cudaCall(arg1,...);
  end = get_time();

  duration = end - begin;
  UPDATE_DATA(CUDA_CALL_ID, duration);

  return ret;
}
```

Figure 2: The anatomy of an IPM library interposition wrapper.

or asynchronous) and kernel launches. Kernel launches are always asynchronous, except when the environment variable `CUDA_LAUNCH_BLOCKING` is set while debugging. A CUDA *stream* is the basic mechanism for concurrent execution of multiple kernels. With the current version (3.1) of the CUDA runtime system, the maximum number of kernels that a device can execute concurrently is sixteen[1].

A. Monitoring and Timing the CUDA Runtime

The CUDA runtime consists of two APIs, the runtime API (e.g., `cudaMalloc()`) and the driver API (e.g. `cuMemAlloc()`). There is a significant overlap in the functionality between these two APIs. The runtime API targets application developers, while the driver API offers a richer set of mechanisms for controlling resource usage and is often preferred by developers of libraries and middleware components.

IPM monitors CUDA applications on the library level by intercepting all runtime and driver API calls. We employ the standard technique of dynamic library interposition [12] to achieve this goal. There are 99 calls in the driver API and 65 calls in the runtime API which are automatically wrapped by IPM's wrapper generator script based on a formal specification file derived from the headers shipped with the CUDA SDK. While all systems we have encountered so far use CUDA through dynamic libraries, it is worthwhile noting that a very similar approach can also be employed for statically linked executables. Specifying the `--wrap foo` option to the linker allows us to provide a wrapper function (named `__wrap_foo` by convention) while making the original call available under the name `__real_foo`. IPM's wrapper generator is flexible enough to generate either variant.

The anatomy of a wrapped CUDA call is shown in Fig. 2. Note that the wrapper allows us to perform actions before and after the actual call is handled by the CUDA runtime. Notably, we can setup *begin* and *end* timers to measure the duration of the call. IPM then stores the event details in its hash table data structure as outlined in Section II.

```
#define REPEAT 10000
__global__ void square(double *a, int N) {
  int i, idx = blockIdx.x;
  for( i=0; i<REPEAT; i++ ) {
    if (idx < N) a[idx] = a[idx] * a[idx];
  }
}

int main() {
  const int N = 100000;
  double *a_h, *a_d;
  size_t size = N*sizeof(double);

  a_h = (double *)malloc(size);
  cudaMalloc((void **) &a_d, size);

  // ... init array, nblocks, blocksz

  cudaMemcpy(a_d, a_h, size,
             cudaMemcpyHostToDevice);

  square <<<nblocks, blocksz>>>(a_d, N);

  cudaMemcpy(a_h, a_d, size,
             cudaMemcpyDeviceToHost);

  cudaFree(a_d); free(a_h);
}
```

Figure 3: A simple CUDA example.

For illustrative purposes, Fig. 3 shows a simple CUDA kernel and host program fragment implementing repeated squaring of double precision floating point numbers passed in an array. Each CUDA thread is used to compute the square of one number in the array. Fig. 4 shows the IPM profile for the invocation for this kernel. Note the large amount of time in `cudaMemcpy` and very little time in `cudaLaunch`[2]. Also note the large amount of time in `cudaMalloc` which, since this is the first call to the CUDA API, actually relates to initialization and setup of the runtime and device.

B. Using the Event API for Device Timing

Except for the duration of blocking memory copies, the host-only timing approach described so far offers only limited insight into the behavior of the GPU. Luckily CUDA offers a mechanism to measure the duration of activities on the GPU. The CUDA event API allows the insertion of events into the execution stream. The status of events can then be investigated and the duration between pairs of events can be computed.

We use a statically allocated *kernel timing table* where we record the start event, the stop event, the stream in which the

[1]NVIDIA CUDA C Programming Guide Version 3.1, Section 3.2.7.3.

[2]We use `cudaLaunch` as an example in the rest of this section. The other functions to launch kernels in the runtime and driver API are handled in a similar way.

```
##IPMv2.0#####################################
#
# command   : ./cuda.ipm
# host      : dirac15
# wallclock : 3.59
#
#                      [time] [count]  <%wall>
# cudaMalloc             2.43       1    67.71
# cudaMemcpy(D2H)        1.16       1    32.24
# cudaMemcpy(H2D)        0.01       1     0.01
# cudaSetupArgument      0.00       2     0.00
# cudaFree               0.00       1     0.00
# cudaLaunch             0.00       1     0.00
# cudaConfigureCall      0.00       1     0.00
#
###############################################
```

Figure 4: IPM's banner profiling report for the kernel shown in Fig. 3.

kernel executes, and a pointer to the kernel function (which is passed as an argument to the `cudaLaunch` call). Our `cudaLaunch` wrapper locates a free slot in the table and stores the stream identifier and kernel pointer and enqueues the start event before the launch and the stop event after the call. After the kernel execution finishes on the GPU, we can query the kernel duration on the GPU using the function `cudaEventElapsedTime`. The only remaining issue is *when* to check for kernel completion. Since kernels are executed asynchronously, we cannot expect the GPU be done with the kernel execution while we are still inside `cudaLaunch` wrapper. It would be possible to check the table for completed operations on each subsequent CUDA runtime call, but doing this too frequently could cause high overheads. Since any data that is used by the main (host) program has to be requested explicitly by a later memory device-to-host memory transfer, it is safe to assume that at least one such memory transfer has to occur after the kernel launch. We therefore chose to check for completion of kernel execution only in memory transfer operations from GPU to CPU. If a completed kernel execution is detected (a `cudaEventQuery` call returns success), the duration is recorded and an entry is placed in the performance data hash table and the occupied slot in the kernel timing table is freed. We use pseudo-function entries of the form `@CUDA_EXEC_STRM00` to denote kernel execution time in our performance hash table. The `@` signals that the entry does not correspond to a host function and STRMi signifies that this execution took place in stream i.

Fig. 5 shows the same kernel execution as before, now including the additional timing from the GPU kernel execution. The banner output provides a GPU execution summary for each stream separately, the XML profiling log contains a per-kernel and per-stream breakdown in addition to that.

```
...
#                      [time] [count]  <%wall>
# cudaMalloc             1.29       1    52.51
# cudaMemcpy(D2H)        1.16       1    47.40
# @CUDA_EXEC_STRM00      1.16       1    47.38
# cudaMemcpy(H2D)        0.01       1     0.01
# cudaFree               0.00       1     0.00
# cudaLaunch             0.00       1     0.00
# cudaSetupArgument      0.00       2     0.00
# cudaConfigureCall      0.00       1     0.00
...
```

Figure 5: IPM profile with GPU kernel timing enabled.

```
...
#                      [time] [count]  <%wall>
# cudaMalloc             1.29       1    52.81
# @CUDA_EXEC_STRM00      1.15       1    47.09
# @CUDA_HOST_IDLE        1.15       1    47.08
# cudaMemcpy(D2H)        0.01       1     0.01
# cudaMemcpy(H2D)        0.01       1     0.01
# cudaLaunch             0.00       1     0.00
# cudaSetupArgument      0.00       2     0.00
# cudaConfigureCall      0.00       1     0.00
...
```

Figure 6: IPM profile with GPU kernel timing and implicit host blocking identification enabled.

C. Measuring Implicit Host Blocking (Host Idle Time) and Identifying Opportunities for Overlap

The previous example provides insight into another important characteristic of GPU kernel execution. The device-to-host transfer is implicitly blocked on the host until the kernel on the GPU completes (compare 0.01 sec. vs. 1.16 sec for the same amount of data transferred).[3] This implicit waiting time in the device-to-host memory transfer is a missed chance to utilize accelerator overlap and thus represents a tuning opportunity.

We set out to quantify this overhead using the following method. First, we identified the set of CUDA operations that exhibit the implicit blocking behavior using a micro-benchmark which exercises each call and compares the timing with a version in which in which we first execute a `cudaStreamSynchronize`. The identified set of calls consists of all versions of synchronous memory related operations, with the notable exception of `cudaMemset` and `cuMemset`. In the IPM wrapper for each identified call we then issue a `cudaStreamSynchronize` for the affected stream and measure how long this operation takes. Then we separately measure the duration of the original call. The implicit waiting time is reported as `@CUDA_HOST_IDLE` in Fig. 6, which is again a pseudo-function entry in IPM's hash table.

To illustrate this in more detail, Fig. 7 shows the IPM monitoring of GPU kernels and host idle identification for

[3]Memory transfer operations are optionally augmented with the direction of the transfer (D2H/H2D) internally by IPM to facilitate this analysis.

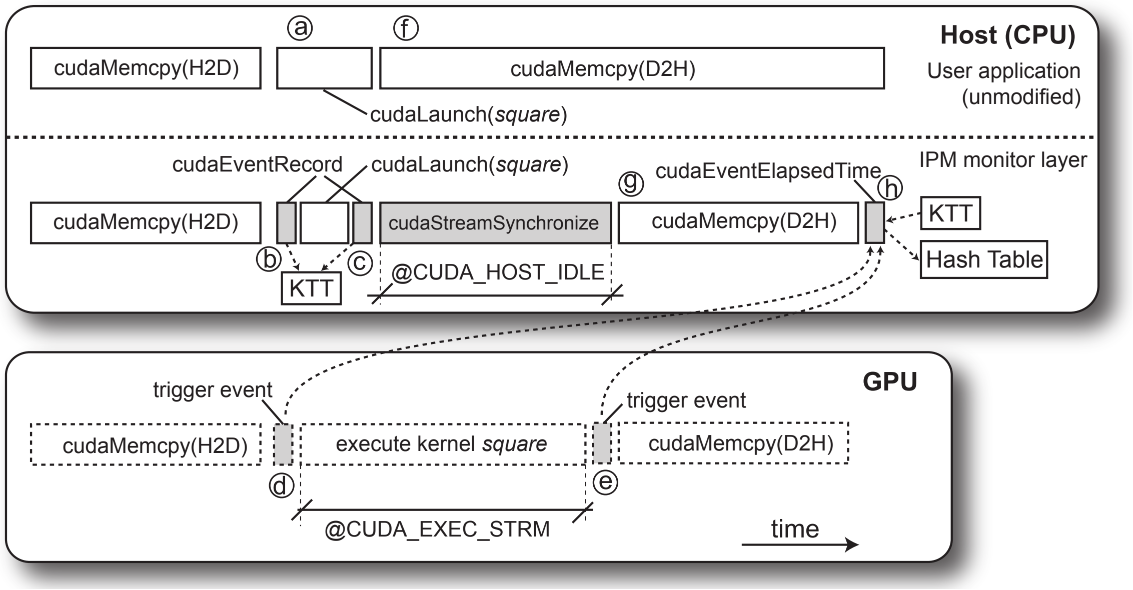


Figure 7: A schematic illustration of the CUDA monitoring approach implemented by IPM. Time proceeds from left to right, the top part of the figure refers to the host while the lower part refers to the GPU, where the activities are only observable by using the CUDA event API.

the preceding example. The figure is organized in three layers with time proceeding from left to right. The top-most layer represents the unmodified user application, the middle layer is the IPM monitoring library linked to the application, and the bottom layer represents the GPU. At ⓐ the kernel `square` is launched. IPM's wrapper for `cudaLaunch` inserts an event before and after the launch (ⓑ, ⓒ), adds an entry into the kernel timing table (KTT), and passes the kernel through to the CUDA runtime. The GPU will execute the kernel function and trigger the events bracketing the kernel (ⓓ, ⓔ). Meanwhile, at ⓕ the user application posts a blocking `cudaMemcpy` immediately after the asynchronous kernel launch. IPM performs host idle identification for this synchronous memory transfer by issuing a `cudaStreamSynchronize` which blocks until the kernel finishes execution at ⓔ. Finally the actual memory transfer occurs at ⓖ. As a last step the kernel timing table is updated with the duration of the executed kernel (ⓗ). For this, the duration of the events triggered in ⓓ and ⓔ is determined using `cudaEventElapsedTime`, the entry in the kernel timing table is freed and performance data is updated in the central hash table.

D. Monitoring Numerical Libraries

To utilize the full potential of supercomputers equipped with GPUs, many applications will have to be restructured and their kernels may have to be rewritten from scratch. On the other hand, many applications today make heavy use of optimized numerical libraries (i.e., FFTW, MKL, or ACML) on conventional multicore CPUs and the switch to an accelerated implementation of these libraries can be a first step to explore the benefits of GPUs. In such cases it is important that performance data can be understood in terms of time spent in such numerical libraries.

NVIDIA ships two optimized numerical libraries with the CUDA runtime (CUFFT and CUBLAS) and we implemented wrappers for both libraries in IPM. There are 13 calls in CUFFT and 167 calls in CUBLAS. Wrappers are again generated automatically from a specification derived from the library's header file and the interposition technique is similar to the one described in Section III-A. In addition to basic timing information, IPM records the size of matrices, vectors, or operations for each call in the *bytes* parameter in IPM's performance data hash table. This allows for a correlation of achieved performance with the size of the operation in later analysis stages.

IV. Evaluation

In this section we evaluate the functionality, accuracy, and overheads of the presented monitoring approach. All experimental results reported here were obtained on the *Dirac* cluster at NERSC. Dirac consists of 48 nodes, each with two Intel Xeon 5530 (Nehalem) quad core processors (8 cores total per node), 24 gigabytes of DDR3 memory, and a single NVIDIA Tesla C2050 ("Fermi") GPU card with 3 gigabytes of device memory. The Dirac nodes are connected via QDR Infiniband. CUDA v3.1 was used in all experiments.

A. GPU Kernel Timing Accuracy

Table I shows an evaluation of IPM's accuracy in measuring the GPU kernel execution time. We selected a number of small benchmarks from the CUDA SDK and compared the timing results obtained from IPM with the data delivered by the CUDA profiler on a single node of the Dirac cluster. The CUDA profiler is activated by setting the environment variable `CUDA_PROFILE` and it writes a trace of kernel execution statistics to a log file. To compare the timings, we sum the kernel execution times over all invocations and compare them with the results measured by IPM.

Evidently, the IPM timing results are in very good agreement with the data delivered by the CUDA profiler. The event-based timing durations used by IPM are always larger than the CUDA profiler results because this technique actually measures the time difference between the events bracketing a kernel, not the kernel itself. The relative difference between the methods is larger for shorter kernels, indicative of a small constant additional overhead as one would expect from this event-based timing. We are currently investigating the improvement of the timing fidelity further by correcting for this overhead but even without correction the results are very satisfactory.

B. Application-Level Runtime Dilatation

In this experiment we tried to evaluate the effective runtime dilatation due to measurement overheads that an application experiences. To account for variations in runtime caused by varying system load, noise and jitter, we performed an ensemble study, repeatedly running the same application with the same inputs, both with and without IPM monitoring enabled.

Fig. 8 shows the histogram of the total runtime observed for 120 runs with and 120 runs without IPM monitoring of the CUDA version of the HPL (High Performance Linpack) [13] version. The mean runtime increased from 126.40 seconds without IPM to 126.67 seconds when monitoring was enabled. This corresponds to an 0.21% increase in runtime and is evidently well below the natural runtime variation between runs. In this experiment IPM monitors all MPI and CUDA events and performs kernel timing and host idle time identification as discussed in III-C. HPL was run on 16 nodes of the Dirac cluster in this setting.

C. CUDA Kernel Monitoring in HPL

Fig. 9 shows a CUBE snapshot of monitoring a CUDA-enabled version of HPL [13] running on 16 nodes of the Dirac cluster. Four kernels are executed on the GPU (`dgemm_nn_e_kernel`, `dgemm_nt_tex_kernel`, `dtrsm_gpu_64_mm`, and `transpose`). Note that this view allows us to observe the distribution of the GPU kernel runtimes on a per-stream basis and across the nodes in the system, permitting the easy identification of imbalances, for example. In the case of HPL, the computation is fairly well balanced and the code is highly tuned for efficiency. `@CUDA_HOST_IDLE` is almost zero, because asynchronous memory transfer operations are used instead of synchronous ones for efficiency by this code. HPL itself then uses the CUDA event API to manually synchronize and it spends a total of between two and five seconds per MPI task in `cudaEventSynchronize` (not shown in Fig. 9). Minimizing this time further would be an opportunity for further performance improvement.

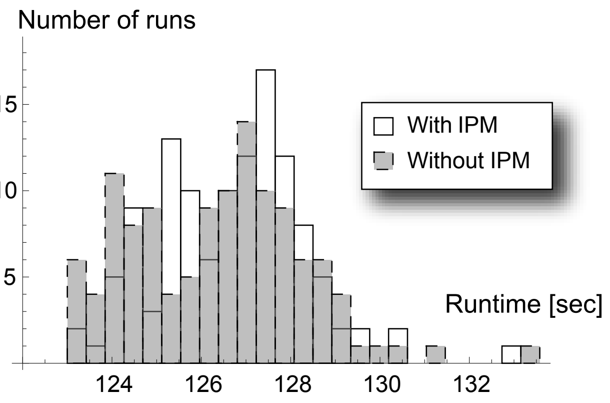


Figure 8: Histogram of execution times of the HPL application, run on 16 nodes of the Dirac cluster (120 runs with IPM and 120 runs without IPM). IPM monitors all MPI and CUDA events, times GPU kernels, and performs host idle identification.

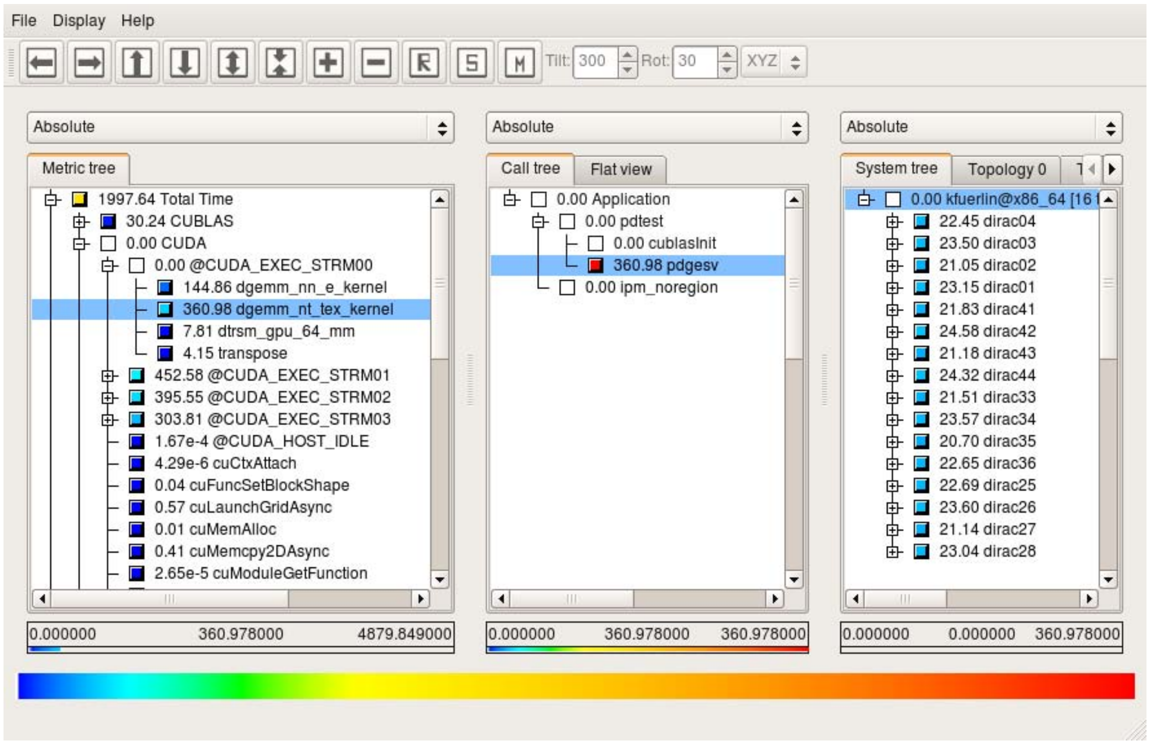
D. PARATEC

PARATEC [14] (PARAllel Total Energy Code) performs ab initio quantum-mechanical Density Functional Theory (DFT) total energy calculations using pseudopotentials and a plane wave basis set. PARATEC is written in Fortran 90 and uses two standard libraries, BLAS and FFTW. In this work we use the NERSC6 version of the code with the medium problem size and we link with CUBLAS to explore the merits of GPU acceleration.

The CUBLAS library can be used from a Fortran code by either linking with *thunking* wrappers or the direct wrappers. The thunking version preserves the usual BLAS calling semantics and implements all interaction with the GPU in the wrapper. Memory on the GPU is allocated, input matrices and vectors are transferred, the actual numerical kernel is invoked, and the results are transferred back to the host. The direct wrappers, on the other hand, just provide the Fortran bindings for the CUBLAS library and memory allocation and transfer have to be done manually within the application. This is clearly less convenient but it does allow for overlap, whereas the thunking version implements purely blocking calling semantics and exposes no opportunity for overlap.

Benchmark	Kernel Invocations	GPU Kernel Execution Time (sec) CUDA Profiler	IPM	Difference (%)
BlackScholes	512	2.540677	2.543700	0.12
FDTD3d	5	0.101354	0.101550	0.19
MersenneTwister	202	1.126475	1.127000	0.05
MonteCarlo	2	0.001988	0.002025	1.87
concurrentKernels	9	0.613755	0.614000	0.04
eigenvalues	300	5.328266	5.331000	0.05
quasirandomGenerator	42	0.039536	0.039736	0.51
scan	3300	1.412912	1.430200	1.22

Table I: Comparing the GPU kernel execution times as reported by IPM with the results obtained by using the CUDA profiler for a number of benchmark examples from the CUDA SDK.



```
1997.64 Total Time
  30.24 CUBLAS
  0.00 CUDA
    0.00 @CUDA_EXEC_STRM00
      144.86 dgemm_nn_e_kernel
      360.98 dgemm_nt_tex_kernel
      7.81 dtrsm_gpu_64_mm
      4.15 transpose
    452.58 @CUDA_EXEC_STRM01
    395.55 @CUDA_EXEC_STRM02
    303.81 @CUDA_EXEC_STRM03
    1.67e-4 @CUDA_HOST_IDLE
    4.29e-6 cuCtxAttach
    0.04 cuFuncSetBlockShape
    0.57 cuLaunchGridAsync
    0.01 cuMemAlloc
    0.41 cuMemcpy2DAsync
    2.65e-5 cuModuleGetFunction
```

```
0.00 Application
  0.00 pdtest
    0.00 cublasInit
    360.98 pdgesv
  0.00 ipm_noregion
```

```
0.00 kfuerlin@x86_64 [16
  22.45 dirac04
  23.50 dirac03
  21.05 dirac02
  23.15 dirac01
  21.83 dirac41
  24.58 dirac42
  21.18 dirac43
  24.32 dirac44
  21.51 dirac33
  23.57 dirac34
  20.70 dirac35
  22.65 dirac36
  22.69 dirac25
  23.60 dirac26
  21.14 dirac27
  23.04 dirac28
```

Figure 9: CUDA and MPI profile of the CUDA-accelerated HPL application. The MPI performance metric hierarchy is below the CUDA metric hierarchy and not visible in this picture.

For an initial study, we linked PARATEC with the thunking CUBLAS wrappers and Fig. 10 shows the scaling of PARATEC on 32 nodes of the Dirac cluster using 32, 64, 128, and 256 MPI processes. Switching from sequential MKL BLAS to CUBLAS accelerates the application by about 35% (the runtime improves from 1976 to 1285 seconds). Fig. 10 shows a breakdown of the wallclock time into time spent in MPI and CUBLAS as well as a further breakdown into the contribution of `MPI_Allreduce`, `MPI_Wait`, and `MPI_Gather` routines as well as `cublasGetMatrix` and `cublasSetMatrix`. The most prominent BLAS routine used by PARATEC is `zgemm` (double complex matrix-matrix multiplication). As mentioned earlier, `cublasSetMatrix` and `cublasGetMatrix` are called by the thunking wrapper and are blocking transfers of input and output matrices, respectively. For this PARATEC input set, the time spent in the transfer dwarfs the time spent in the actual `zgemm` computation. Overall, PARATEC scales well up to 128 processors, then MPI starts to dominate, particularly the contribution of `MPI_Gather` becomes very large. We are currently investigating the exact cause for this behavior but we assume that it is caused by NUMA effects. The time spent in CUBLAS remains relatively constants. While multiple MPI processes share a single GPU, but the dataset is also reduced as we increase the number of MPI processes.

We plan to refine these initial PARATEC results by exploring opportunities for overlap and simultaneous use of the CPU+GPU BLAS routines. The comparatively large amount of time spent in the synchronous memory transfer

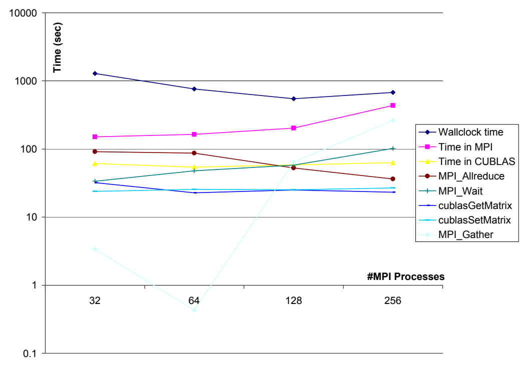


Figure 10: The scaling of PARATEC.

operations that we were able to identify with the help of IPM suggests that this could be very beneficial.

E. Amber

Amber [15] is a molecular dynamics package for the simulation of biomolecules. In this work we use a pre-release of the CUDA version of the PMEMD code that runs on multiple GPUs simultaneously using MPI for communication [16], [17]. The test case is the Joint AMBER/Charmm (JAC) DHFR benchmark which consists of the protein DHFR solvated with TIP3 water molecules. There are a total of 23,558 atoms in the simulation and we run for 10,000 timesteps.

Fig. 11 shows the profile of Amber executed on 16 nodes of the Dirac cluster. There are 39 GPU kernels (not shown in Fig. 11 but included in the full XML profiling log) and Amber also uses the CUFFT library. The most time-consuming kernels are (in decreasing order of contribution to the execution time) `CalculatePMEOrthogonalNonbondForces` (ca. 37% of GPU time), `ReduceForces` (18%), `PMEShake` (10%), `ClearForces` (8%) sec, and `PMEUpdate` (7%), the rest of the kernels contribute about 20% of GPU time. Amber achieves a quite high GPU utilization (35.96% of total wallclock execution time) and despite using synchronous memory transfer operations, the host idle time is very small (0.08%). However, the code also spends a lot of time (22.50%) in `cudaThreadSynchronize` (host-side) waiting for the kernels on the GPU to finish. As an opportunity for further optimization, in a fully heterogeneous implementation, the CPU could instead be utilized for computation as well, increasing overall performance. The `PMEShake` and `PMEUpdate` kernels are very

```
##IPM##########################################################
#
# command   : pmemd.cuda.MPI -O -i mdin -c inpcrd.equil...
# start     : Tue Sep 28 12:35:09 2010  host      : dirac18
# stop      : Tue Sep 28 12:35:55 2010  wallclock: 45.78
# mpi_tasks : 16 on 16 nodes            %comm     : 0.60
# mem [GB]  : 4.41                      gflop/sec: 0.00
#
#           :     [total]      <avg>        min         max
# wallclock :      732.10      45.76      45.73       45.78
# MPI       :        4.39       0.27       0.09        0.31
# CUDA      :      479.71      29.98      27.73       33.80
# CUFFT     :        0.87       0.05       0.00        0.86
# %wall     :
#  MPI      :                   0.60       0.20        0.68
#   CUDA    :                   4.62      73.85       73.85
#   CUFFT   :                   0.12       1.88        1.88
# #calls    :
#   MPI     :        1326         82         75         201
# mem [GB]  :        4.41       0.28       0.24        0.31
#
#                         [time]      [count]      <%wall>
# @CUDA_EXEC_STRM00       263.27      1927994        35.96
# cudaThreadSynchroni     164.76      1246794        22.50
# cudaMemcpyToSymbol       17.19       276000         2.35
# cudaGetDeviceCount       16.72           32         2.28
# cudaLaunch                9.48      1927994         1.30
# cudaMemcpy                4.17       335607         0.57
# MPI_Bcast                 3.71          816         0.51
# cudaConfigureCall         1.69      1927994         0.23
# cudaSetupArgument         1.16      1937824         0.16
# @CUDA_HOST_IDLE           0.62       335943         0.08
# MPI_Allgatherv            0.51           16         0.07
# cufftExecR2C              0.44        10000         0.06
# cufftExecC2R              0.42        10000         0.06
# cudaGetLastError          0.19      1706778         0.03
```

Figure 11: Profile of Amber executed on 16 nodes of the Dirac cluster.

well load balanced (across the 16 MPI processes), the `CalculatePMEOrthogonalNonbondForces` kernel is reasonably load balanced as well, while `ReduceForces` and `ClearForces` show imbalances of up to a factor of 55%, an elimination of which is a potential avenue for further optimization.

V. Related Work

There are several tools available to monitor and analyze the performance of CUDA programs. NVIDIA Parallel Nsight is a CUDA development plug in for Microsoft Visual Studio and with debugging and performance monitoring (profiling and tracing) capabilities, only available for Windows platforms. CUDA Profiler and CUDA Visual Profiler are provided by NVIDIA and are available for several operating systems. Both tools record the duration of individual kernel invocations and allow the measurement of GPU performance counters. CUDA Visual profiler has a graphical user interface and will run a program multiple times if this is required due to restrictions on simultaneously countable event sets, while the CUDA profiler is a CLI tool that writes a text-based profiling report to a file. Both tools are well suited to optimize individual CUDA kernels in a workstation-type setting but have two major drawbacks for the usage in large-scale GPU cluster environments. First, they lack support for monitoring other widely used parallel programming approaches such as OpenMP and MPI and thus can't provide a holistic picture of application behavior in terms of these factors. Secondly, they are not meant to be used on large-scale machines and provide no mechanism to integrate a performance data across nodes. A user would have to manually combine and integrate several data files (one for each MPI process) – a tedious and error prone process. In comparison, IPM already supports the monitoring of MPI, OpenMP and file-IO and with this work we added support for CUDA and accelerated numerical libraries.

There have also been forays into monitoring accelerators by established scientific computing performance tools. The tracing tool Vampir and the associated tracing library vampirtrace have recently been extended with support for CUDA and OpenCL [18]. This `LD_PRELOAD`-based monitoring approach is similar to ours, but the intended usage of the monitored data is different. While Vampir is a commercial tracing tool designed for the detailed manual analysis of time-stamped event logs at moderate scale, IPM is a freely available highly scalable profiling tool intended for quantifying resource consumption and high-level application metrics. IPM monitors the full set calls in the CUDA runtime and driver API as well as CUBLAS and CUFFT usage to get a comprehensive view of accelerator usage, while Vampir appears to be presently limited to the runtime API. Both IPM and Vampir derive utilization metrics for the GPU and quantify missed opportunities for accelerator overlap.

The developers of TAU (tuning and analysis utilities) have also been working on providing support for CUDA for their tool set. In [19] a technique for timing GPU kernel execution using the event API similar to ours is presented. However, this approach seems to require changes to the user's source code ([20], Related Work section) to implement the kernel timing, while IPM's kernel timing approach is entirely implemented in the dynamically linked library and no modifications or even a re-compilation of the application is needed. In [20] an experimental CUDA driver is utilized to allow a more detailed analysis by providing callbacks that a tool can register for. Unfortunately, the presented techniques are not applicable to the publicly released drivers used in production environments that lack the callback functionality. Neither approach is available in the released versions of TAU.

A paper by Du et al. [21] provides a study of tool usage in the context of the development of numeric linear algebra kernels. The authors highlight the importance of monitoring the interplay of multiple kernels and their integration with host-side execution environment in addition to tuning individual kernels with the CUDA profiler. The absence of an integrated monitoring solution required them to manually instrument the CUDA runtime calls and made analysis unnecessarily tedious, a situation that we tried to address with the extensions to IPM presented in this paper.

VI. Conclusion and Outlook

We have presented our comprehensive monitoring solution for GPU-enabled clusters and supercomputers for the CUDA programming model and accelerator libraries. We have extended the basic host-side timing of CUDA runtime calls with a mechanism that allows the timing of kernels on the GPU on a per-kernel and per-stream basis. By identifying and quantifying implicit waiting times in blocking memory transfers we have included a metric that captures missed accelerator overlap.

We have shown that our approach is very efficient and achieves high monitoring accuracy. In previous work [9] have demonstrated IPM's scalability up to several tens of thousands of cores on modern supercomputers. Although we presently only have access to a small size GPU cluster, the enhancements to IPM presented in this paper will be immediately applicable to large scale GPU-equipped systems as well, since IPM's scalability is not impacted by the monitoring of the CUDA events. We believe the true benefit of comprehensive accelerator monitoring will in fact become more apparent as the size of the GPU machine increases and the focus has to shift from isolated kernel tuning to holistic performance assessment taking into account the plethora of performance-influencing factors.

Future work is planned along several directions. First, the integration of GPU hardware performance counters would be useful for gaining more insight into kernel behavior than is possible from timing information only. Unfortunately there is currently no documented interface to access the counters but we expect such an interface to become available either through PAPI [22] or a published interface provided by NVIDIA. IPM already supports Component PAPI [23] and it would thus be easy to leverage a GPU counter component. Second, while our present work focused on CUDA, the library-based interposition monitoring technique is similarly applicable to OpenCL. Third, we are working on using the derived monitoring data for performance modeling and advanced guidance to users on the merits or pitfalls of accelerating their applications.

Acknowledgment

This work was supported in part by the NSF under award OCI-0721397. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. The authors would like to thank Bryan Catanzaro for his insightful feedback and suggestions for improvement and Ross Walker for providing access to the AMBER code.

References

[1] Cray and NVIDIA, "Press release: Cray to add NVIDIA GPUs to the Cray XE6 Supercomputer." Seattle, WA and San Jose, CA, September 21, 2010.

[2] CULAtools, "The CULA web page, http://www.culatools.com/."

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *Journal of Physics: Conference Series*, vol. Vol. 180.

[4] M. Wolfe, "Implementing the PGI accelerator model," in *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. New York, NY, USA: ACM, 2010, pp. 43–50.

[5] The Portland Group, Inc., "The PGI Fortran and C accelerator programming model," available at http://www.pgroup.com/accelerate.

[6] CAPS, "HMPP workbench http://www.caps-entreprise.com/hmpp/."

[7] D. Skinner, "Performance monitoring of parallel scientific applications," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-PUB-5503, May 2005.

[8] K. Fürlinger, N. J. Wright, and D. Skinner, "Performance analysis and workload characterization with IPM," in *Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing*, Dresden, September 2009.

[9] ——, "Effective performance measurement at Petascale using IPM," in *Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS 2010)*, Shanghai, China, Dec. 2010.

[10] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, "Scalable parallel trace-based performance analysis," in *Proceedings of the 13th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2006)*, Bonn, Germany, 2006, pp. 303–312.

[11] FZ Juelich, "CUBE webpage," http://www.fz-juelich.de/jsc/scalasca/software/download.

[12] G. Nakhimovsky, "Debugging and performance tuning with library interposers," Jul. 2001, http://developers.sun.com/solaris/articles/lib_interposers.html. Retrieved 2010/03/01.

[13] M. Fatica, "Accelerating Linpack with CUDA on heterogenous clusters," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 46–51.

[14] "The PARATEC web page, http://www.nersc.gov/projects/paratec/."

[15] "Amber home page http://ambermd.org."

[16] S. L. Grand, A. W. Goetz, D. Xu, D. Poole, and R. C. Walker, "Acceleration of Amber generalized born calculations using NVIDIA graphics processing units." 2010, in preparation.

[17] ——, "Achieving high performance in Amber PME simulations using graphics processing units without compromising accuracy," 2010, in preparation.

[18] R. Dietrich, T. Ilsche, and G. Juckeland, "Non-intrusive performance analysis of parallel hardware accelerated applications on hybrid arichtectures," in *Proceedings of the 1st International Workshop on Software Tools and Tool Infrastructures (PSTI 2010), co-located with ICCP 2010*, San Diego, CA, Sep. 2010.

[19] S. Mayanglambam, A. D. Malony, and M. J. Sottile, "Performance measurement of applications with GPU acceleration using CUDA," in *Proceedings of the International Conference on Parallel Computing (ParCo)*, Sep. 2009.

[20] A. D. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam, "An experimental approach to performance measurement of heterogeneous parallel applications using cuda," in *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*. New York, NY, USA: ACM, 2010, pp. 127–136.

[21] P. Du, P. Luszczek, S. Tomov, and J. Dongarra, "Mixed-tool performance analysis on hybrid architectures," in *Proceedings of the 1st International Workshop on Software Tools and Tool Infrastructures (PSTI 2010), co-located with ICCP 2010*, San Diego, CA, Sep. 2010.

[22] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. J. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, 2000.

[23] "Component PAPI documentation: http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI-C.html."