Eurographics Conference on Visualization (EuroVis) 2013
B. Preim, P. Rheingans, and H. Theisel
(Guest Editors)

Volume 32 (2013), Number 3

A Visual Approach to Investigating Shared and Global Memory Behavior of CUDA Kernels

Paul Rosen†

Scientific Computing and Imaging Institute
University of Utah

Abstract

We present an approach to investigate the memory behavior of a parallel kernel executing on thousands of threads simultaneously within the CUDA architecture. Our top-down approach allows for quickly identifying any significant differences between the execution of the many blocks and warps. As interesting warps are identified, we allow further investigation of memory behavior by visualizing the shared memory bank conflicts and global memory coalescence, first with an overview of a single warp with many operations and, subsequently, with a detailed view of a single warp and a single operation. We demonstrate the strength of our approach in the context of a parallel matrix transpose kernel and a parallel 1D Haar Wavelet transform kernel.

Categories and Subject Descriptors (according to ACM CCS): Hardware [B.8.2]: Performance and Reliability—Performance Analysis and Design Aids

1. Introduction

The explosion of high-performance computer hardware over the recent past has been a boon for computational science, engineering, and many other fields. Scientists and engineers are now able to process increasingly complicated phenomena with ever greater level-of-detail. However, the move to high-performance software has not been an easy one. The complexity of hardware makes programming scalable algorithms challenging. A significant investment of time and highly tuned experience and intuition are required to optimize algorithms. This is in large part due to the lack of good profiling and visualization tools.

We present a new approach for investigating the interaction of hardware and software for highly parallel environments, CUDA in particular. We focus our efforts on the behavior of the two most heavily used memory systems within CUDA, global and shared memory. Our visualization uses multi-level comparative and qualitative analysis stages to isolate representative behavior and identify problematic portions of execution. As a whole, this approach gives a top-down view of execution for identifying *where*, then understanding *why* memory performance might struggle.

At the top level, the full execution is visualized and a comparative analysis is performed. Here, users can identify groups of threads whose execution is most representative and those whose execution most deviates from the representative behavior. As representative warps are selected, the user proceeds to lower level qualitative visualizations. At these levels the visual interface shows detailed views of memory access for smaller groups of threads permitting the identification of the source of performance bottlenecks.

2. Background

Debugging tools are a crucial component of the software development work cycle. These tools, such as GNU Debugger (GDB), Microsoft Visual Studio, NVIDIA Nsight [NVI12c] and TotalView Debugger [Tot10], provide a transparent view of an application, enabling detailed inspection of execution. The machine-oriented level at which these tools operate assists in searching out the sources of errors. However, these tools are only focused on *correctness*. In contrast, the work we present assumes correctness and is only interested in enhancing *performance*.

Profiling is the approach most frequently used to understand the complex performance interactions of software and hardware. Profilers observe an application's execution through specialized hardware counters that collect data such as processor utilization, cache misses, etc. Tools such as PAPI [TJYD09] and Valgrind [NS03] provide convenient high-level APIs for accessing these counters which are otherwise only accessible directly through low-level interfaces.

† e-mail: prosen@sci.utah.edu

© 2013 The Author(s)
Computer Graphics Forum © 2013 The Eurographics Association and Blackwell Publishing Ltd. Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

Desktop profilers, such as GNU GProf [GNU12], Intel VTune [Int12], and Apple Instruments [App12], use these performance counters to gather statistics and communicate those values to a software developer. For graphics processors, similar statistics are collected through the NVIDIA Visual Profiler [NVI12d] (Figure 1) and NVIDIA Nsight [NVI12c]. These products remain the work most closely related work to ours, though they operate at critically different granularity.

For supercomputers, VampirTrace [NAW*96, Zen12] and TAU [SM06, Uni12] provide the same statistics as desktop profilers for each node, along with statistics for the entire cluster. Recent work [SLB*11] visualizes this data directly in the application domain, better connecting the performance to the data. Visualization has also been used for a variety of other performance related tasks including memory allocation [MT07] and scheduling [BBH08]. These techniques are more or less outside of the scope of this work and, in some cases, complement our approach. Most of these profiling tools work at coarse granularity, limiting potential insights to large scale bottlenecks. The recent work of Choudhury [rICPP08, rICR11] has focused on individual memory transactions. While their approaches enable new insights, they have not yet been shown to scale to large memory traces or parallel applications the way this work does.

3. An Overview of the CUDA Architecture

The model for CUDA is to execute a single *kernel* of code using thousands of threads on a *grid*. The threads are all grouped into *blocks* of execution, from which groups of 32 threads form a *warp*. For some operations, warps can be divided into *half-* and *quarter-warps* for servicing. Operation on the threads of a warp are executed simultaneously in SIMD (single-instruction, multiple-data) fashion, and due in large part to hardware optimizations, the behavior of all threads in the warp affect the performance of a single thread. Multiple warps from different blocks can execute simultaneously on different processors, giving an overall MIMD (multiple-instruction, multiple-data) behavior.

This work focuses on the two most heavily used components of the CUDA memory hierarchy, shared and global memory. *Global memory* or *device memory* is a large, slow to access memory that is read/write accessible to all threads, warps, and blocks. *Shared memory* is a small, fast on-chip user managed read/write cache, only accessible to the threads and warps from the same block. Shared memory is an n-way associative memory[†] with each *bank* able to service only one memory address at a time.

In addition to global and shared memory, other memory components exist in the CUDA architecture. Local memory usage is the result of local array storage and register spillage, accessible to an individual thread only, and stored in global memory. Constant memory is read-only memory initially stored in global memory, but once used, it is held in a special on-chip cache. Finally, surface and texture memory use specialized hardware to quickly access and interpolate data from global memory when memory accesses have good 2D spatial locality. For a more detailed explanation of these systems, see the NVIDIA CUDA Programming Guide [NVI12b].

3.1. Performance Optimization

There are three main approaches to optimizing the execution of CUDA kernels [NVI12b]. The first is to maximize utilization by exploiting the optimal amount of parallelization, minimizing the idle time of the compute device and the buses connecting the device and host. The second is to maximize memory throughput by minimizing the number of memory accesses and optimizing the memory access patterns of kernels. This optimization is the main thrust of this work. The final optimization approach is to maximize instruction throughput by minimizing control flow, synchronization, and high-cost arithmetic instructions.

The first approach to optimizing memory performance is to reduce the number of global memory accesses. This is most often done by offloading repeated global memory accesses to shared memory (or one of the other memory subsystems).[‡] For global memory operations which cannot be avoided, the remaining performance optimization is to use *coalesced accesses*. When the threads of a warp access memory in a friendly pattern, the hardware attempts to coalesce those memory accesses into fewer, more efficient memory transactions. Poorly formulated accesses will issue many transactions, and, due to limitations on the smallest addressable unit, the transaction may end up underutilized.

For shared memory, performance is penalized by *bank conflicts*. Bank conflicts are produced when multiple threads in a warp request different addresses from the same bank. Accesses are serialized into multiple transactions, since each bank is limited to one address per transaction.

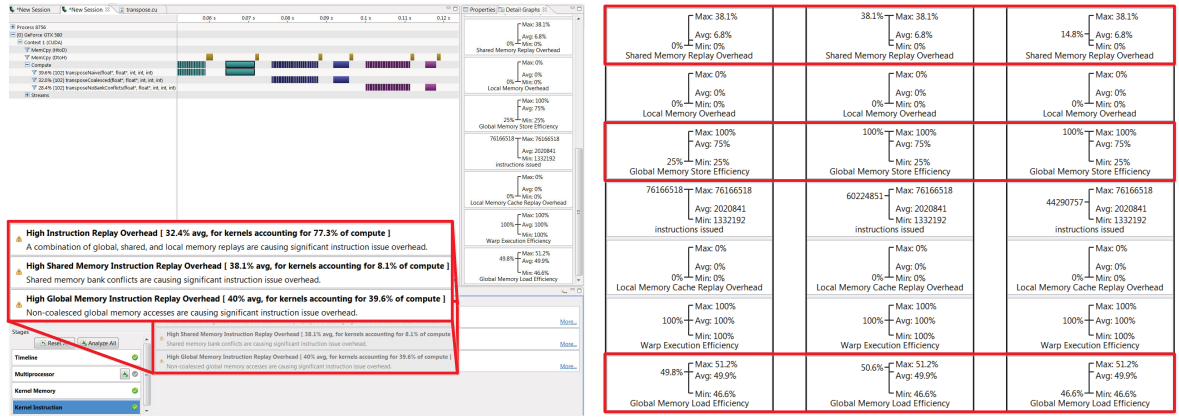
3.2. Performance Data

CUDA devices, like most computer hardware, contain a number of built-in performance counters. These performance counters are available at relatively low granularity with no built in ability to capture the detailed data we seek. Instead, we look towards emulation using gpuocelot [DKYC10, KDY11]. Gpuocelot is a dynamic compilation framework for PTX programs that allows executing the program in a wide variety of environments, including emulated, GPU, and CPU environments. In emulation mode, gpuocelot provides tracing functionality for capturing thread, warp, and block activities, including instruction type, memory addresses, and active thread flags, for every instruction. The trace only provides a list of logical activities with no information on hardware effects. A hardware simulation step is then used to approximate hardware behaviors, such as shared memory bank conflicts and global memory coalescence. This

[†] CUDA 1.x has 16 shared memory banks, and CUDA 2.x onward has 32 shared memory banks.

[‡] In newer architectures, global access is also sped up by the inclusion of caching hardware which supports a user selectable combination of L1 and L2 cache. Our analysis excludes this hardware.

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.



(a) Overview of the user interface

(b) Information pertaining to each of 3 implementations

Figure 1: *NVIDIA Visual Profiler output for matrix transpose identifies existence of errors (marked in red) but not the source.*

is done via a custom component modeled on information provided by NVIDIA [NVI12b].

4. Performance Analysis Approach

One critical component to our approach is fully understanding how we envision this tool fitting into existing software development workflows. As software engineers complete development and correctness checking, they usually move to performance optimization (obviously there is some iteration between these stages). We envision first beginning the process with existing tools such as NVIDIA Visual Profiler (Figure 1). These tools are good at identifying the existence of performance bottlenecks (even down to the line of source code causing global memory slowdowns), but fail to give the visual insights as to exactly *why* these operations cause slowdowns. These details include which warps may be causing the slowing down, the order of memory access, and the shape of resulting memory transactions. For example, the columns of Figure 1(b) show various shared and global memory bottlenecks for 3 different versions of matrix transpose, however they give no insight as to the exact cause (which will be discussed in Section 6.1). Once the existence of a bottleneck is confirmed, we envision our tool being used.

Coming to our tool, the first step is a comparative analysis used to reduce the number of warps to a subset of *representative warps*. This step compares the performance of each warp to that of every other warp. The idea being, if many warps have similar performance, you only need to do further investigations on one of them, not all of them.

Next is a qualitative analysis of the representative warps to identify operations that are performance bottlenecks. The goal of this step is to select the global operations failing to sufficiently coalesce and the shared operations producing many conflicts. Once familiar with our visual metaphors, we believe this is easily accomplished in our approach.

Finally, software engineers are able to investigate each individual bottleneck operation. At this level, we provide as much information as possible about the layout of the memory transactions. However, ultimately it remains the responsibility of the software engineer to optimize their software.

5. Memory Behavior Visualization

We have developed an approach to exploring the behavior of hundreds of thousands of simultaneously executing threads. Our top-down approach works by first using a *comparative analysis* to identifying warps whose behavior is representative, followed by a *qualitative analysis* of those warps to ultimately reveal the source of performance bottlenecks. This style of approach will play a critical role in exploration of many type of large data, not just CUDA memory traces.

5.1. Comparative Analysis

At the top-most level, the *comparative analysis* stages were designed as an overview of all threads over all time. For such a view is it critical to highlight any data element which is *potentially* interesting for user exploration. Our approach focuses on identifying representative and outlier behavior, leaving understanding of the performance quality for lower level investigations. Therefore, we decided to use a visualization metaphor which highlights differences, indicating to the user candidates for exploration. This type of filtering is enabled through a two-level process, closely coupled to the underlying CUDA architecture, for isolating individual warps of interest.

5.1.1. Comparing Operation on Warps

Understanding the behavior on the scale of thousands of threads requires formulating derivative metrics for comparing large groups of threads. We focused on establishing metrics for comparing warp behavior.

When comparing an operation on two warps, A and B, there are 4 aspects which are interesting for our comparison purposes: instruction type, active threads, and shared and global memory access.

The difference between two instructions, A_{ins} and B_{ins} is a simple comparison of type.

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

$$d_{ins}(A,B) = \begin{cases} 0 & \text{if } A_{ins} = B_{ins} \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

The comparison of active threads considers A_{thrd} and B_{thrd} to be the set of threads which are currently active in each warp. By taking the intersection of those sets, active threads which are common to both are found. Finally, the cardinality gives the number of commonly active threads.

$$d_{thrd}(A,B) = 32 - |A_{thrd} \cap B_{thrd}| \quad (2)$$

For global memory, both the size and shape of memory access is important to performance. Therefore, our comparison is the sum of pairwise differences of memory transaction sizes, where $A_{glob}(i)$ and $B_{glob}(i)$ represent one memory transaction from each. In this way, warps with a different number of transactions, as well as those whose transactions are of different sizes, will both be highlighted.

$$d_{glob}(A,B) = \sum |size(A_{glob}(i)) - size(B_{glob}(i))| \quad (3)$$

The most important feature of shared memory access is the number of bank conflicts which occur. So, we consider the difference between the number of bank conflicts A_{conf} and B_{conf}. We decided not to account for bank access order, since order does not directly affect performance the way that the number of serialized accesses does.

$$d_{conf}(A,B) = |A_{conf} - B_{conf}| \quad (4)$$

The final difference between two warps is found by taking the scaled sum of differences.

$$\begin{aligned} d_{Warp}(A,B) = {} & s_{ins} \cdot d_{ins}(A,B) + s_{thrd} \cdot d_{thrd}(A,B) \\ & + s_{glob} \cdot d_{glob}(A,B) + s_{conf} \cdot d_{conf}(A,B) \end{aligned} \quad (5)$$

To balance the impact of any component, all differences are weighted. For all experiments and figures the fixed value weights were selected as follows. The instruction difference is left unchanged, $s_{ins} = 1$. Thread difference was selected as $s_{thrd} = 1/32$, based upon the number of threads in a warp. Shared and global memory are weighted in a more ad-hoc way. Our selection of $s_{conf} = 1/8$ indicates an 8-way conflict, which is a fairly serious conflict. Global scaling was selected as $s_{glob} = 1/1024$, since the largest transactable memory is 1024 bytes. For color mapping, the value of d_{Warp} was always clamped to $[0,1]$. When considering multiple operations, the mean of differences for all operations is considered.

5.1.2. Grid-Level Visualization

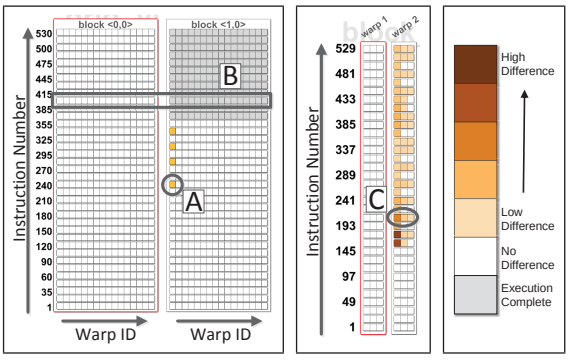
The top visual interface is the grid-level visualization which encodes a representation of every thread used to execute the kernel over long time sequences. The visual interface, as shown in Figure 2(a), was laid out in a manner consistent with the architectural organization of CUDA. Each block of the execution is placed into its own individual region. A representative block, colored in red, is selected by the user. To highlight variations between blocks, a simple metaphor of an indicator light (Label A) was used. Each column of indicator lights represents the behavior of one warp while each row encompasses multiple operations (Label B), the number of which is user adjustable via scrolling.

To color the indicators, a pairwise comparison between each warp in the selected representative block and the current block is performed using Equation 5. The indicator lights are colored with a sequential color map shown in Figure 2(c). When a block terminates earlier, those associated indicator lights are colored gray (Label B). In this way, a warp-by-warp comparison is performed between blocks, indicating blocks that perform similarly to the selected block and those who deviate significantly from the selected block. In Figure 2(a) Label A, the yellow-orange color indicates that warp #1 of $block(1,0)$ deviates from warp #1 of the representative block, $block(0,0)$. It does not however indicate whether quality is good or bad in either.

5.1.3. Block-Level Visualization

After a representative block is selected, a block-level visualization, as seen in Figure 2(b), is displayed. The block-level visualization, being a second filtering pass, should maintain as closely as possible the visual metaphors used in the grid-level visualization and be organized in a manner consistent with the CUDA architecture.

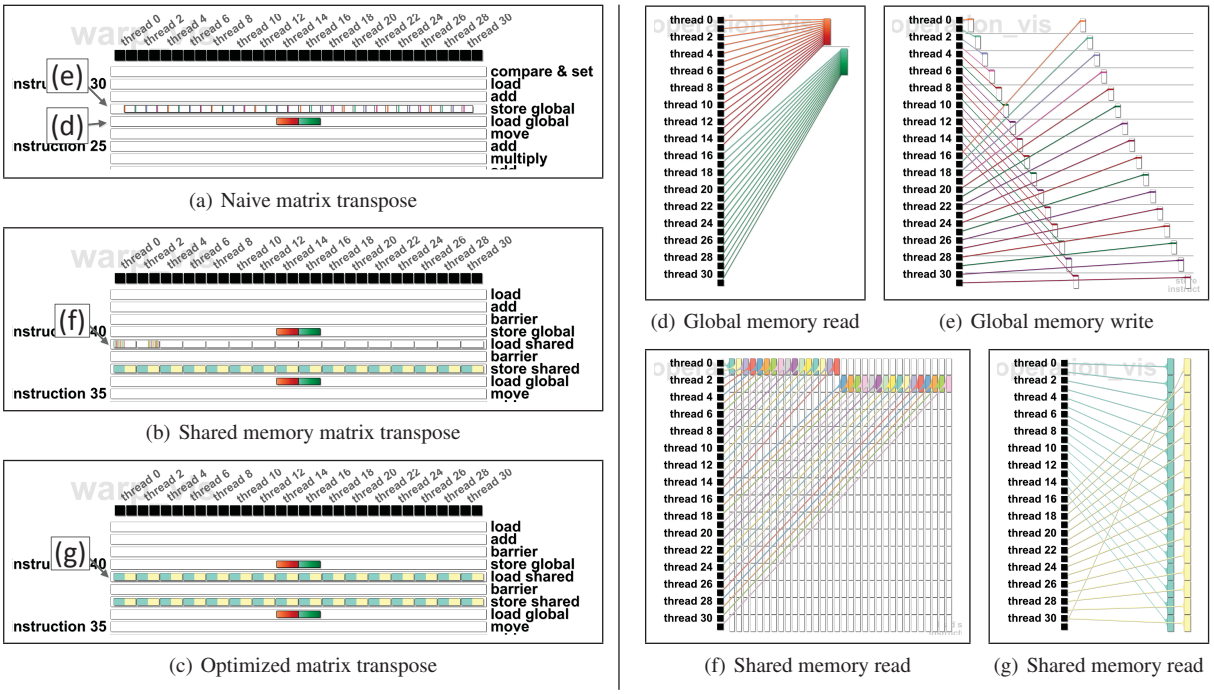
In the block-level visualization, each warp associated with the block is given its own region with each region containing a set of indicator lights. The rows of indicator lights still represent a collection of multiple operations. However, now each of the three columns represent a different comparison metric (Label C). The first column represents difference in active threads using Equation 2; the second column represents differences in shared memory access using Equation 4; and the final column represents the differences in global memory access using Equation 3. In a similar fashion to the grid-level visualization, all warps are compared to the representative warp selected in red and differences are colored the same sequential color map as in the grid visualization (Figure 2(c)). In Figure 2(b) Label C, the lights indicate that during this set



(a) Grid visualization (b) Block vis. (c) Color map

Figure 2: *Breakdown of grid and block visualizations where: A is an indicator light highlighting difference; B represents indicator lights for block-to-block comparison; and C represents indicator lights for warp-to-warp comparison.*

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.



(a) Naive matrix transpose

(b) Shared memory matrix transpose

(c) Optimized matrix transpose

(d) Global memory read

(e) Global memory write

(f) Shared memory read

(g) Shared memory read

Figure 3: *Warp visualizations for three version of matrix transpose (left) and four operation visualizations associated with the various versions of matrix transpose (right). The implementations show a mix of good quality global (d) and shared (g) transactions, as well as poor quality global (e) and shared (f) transactions.*

of operations, warp #2 varies from the representative (warp #1) in terms of threads, shared memory, and global memory. Again, no indication of quality is made.

5.2. Qualitative Analysis

Once representative and outlier warps have been isolated, investigations of quality can be pursued. The *qualitative analysis* stages are designed to first identify operations which may be causing negative side-effects to performance. Then, descriptive access is given to the configuration of those transactions. The hope being, detailed information about the associated memory access pattern will enable developers to potentially reorient the memory access of their kernels.

5.2.1. Warp-Level Visualization

The next level of investigation visualizes the activity of a single warp over multiple operations. The visualization, as shown in Figures 3(a) and 3(b), shows the series of operations and contains 4 important types of visual component. The visual metaphors used for these components are designed to mimic the metaphors for the final level of investigation. Therefore, their motivation will be discussed in Section 5.2.2.

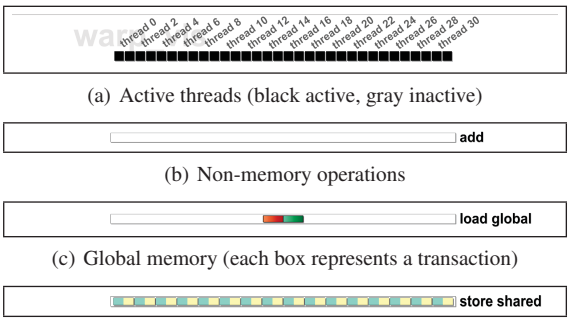
The first component is the thread interface (Figure 4(a)). Each thread of the warp is represented with a box. During operations where the thread is active, the box is colored black, otherwise it is gray. The size of the box is used as a temporal indicator, varying it by the percentage of time a thread was active over the prior 100 instructions.

The next visual component represents operations which do not access global or shared memory (Figure 4(b)). These operations are represented by an empty box. The boxes and their associated instruction text are colored black if the warp is active, or gray if the warp is inactive, during these operations.

Global memory operations (Figure 4(c)) are represented by a series of small boxes, one for each memory transaction. The boxes are sized relative to the size of the memory transaction. The colors are assigned categorically, where each category represents a continuous address space. In Figure 4(c), red and green indicate 2 continuous address spaces. Further, only potions of the memory transaction which are used by one or more of the threads are colored (see Figures 8(a) and 8(c) for examples of underutilized transactions). A good global memory access will have few solid colored boxes (such as Figure 3(a) Label (d)), while a bad memory access will contain many mostly empty boxes (such as Figure 3(a) Label (e)).

Shared memory access (Figure 4(d)) is similarly represented by a series of boxes. In this case, each box represents one bank of the shared memory. To color the boxes, each bank is subdivided by the number of memory transactions which occur during the operation. Each transaction is assigned a color, and the banks used by the transaction are colored accordingly. Therefore, a solid color or low frequency color change indicates a good memory access pattern (such as those in Figure 3(c) Label (g)), and high frequency color or many empty banks indicate a bad memory access pattern (such as those in Figure 3(b) Label (f)).

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.



(a) Active threads (black active, gray inactive)

(b) Non-memory operations

(c) Global memory (each box represents a transaction)

(d) Shared memory (each box represents a bank, while each color represents a transaction)

Figure 4: *The 4 metaphors used in the warp visualization.*

5.2.2. Operation-Level Visualization

Finally, we enable detailed investigation at the single warp, single operation level (Figures 5). The metaphors used in this interface is tightly coupled to those used at the warp-level. Here, the threads associated with the warp are laid out along the left side and colored according to their active status, just as in the warp-level visualization. For operations without any memory access, the interface is complete.

The next scenario is a global memory access as seen in Figure 5(a). Here, the number of transactions and how those transactions connected to their issuing threads is critical. Each memory transaction is drawn in its own column. The size of the memory transaction is indicated by the size of the box. For example, the memory transaction indicated by Label B is larger than memory transaction indicated by Label C in the same operation. Each box is initially drawn empty and any pieces of that memory transaction that are used by the threads are colored and connected to the thread accessing it. In this way, a rough sense of utilization is quickly accessible. For example, Label C is only partially filled, indicating a large transaction with minimal usage. The boxes are laid out vertically as rough indication of memory address. Assuming the memory accessed is continuous, the layout is continuous. If, however, there is a jump in the address space, a spacer is added, the color is changed, and the new address space is continued (see Label A). For global memory access, good quality operations will have high utilization with a small number of memory transactions. Too many memory transactions flood the memory controller, and underutilization floods the memory bus with unused data.

The final scenario is a shared memory access as shown in Figure 5(b). The motivation for the design is similar to that of global access. The most important components of the operation are the number of transactions and their connection to issuing threads. The rows of the visualization are the various banks, each represented by a box (see Label D). Again, each serialized transaction to shared memory is shown as a column (i.e. more columns, more bank conflicts), with each thread connected to its particular access. Each column is also colored uniquely to improve the differentiation of the various lines connecting threads and banks. Bank accesses which are broadcast to multiple threads are indicated by a star within the bank. For shared memory, the most important aspect to performance is minimizing the number of serialized accesses.

5.3. Correlation to Source Code

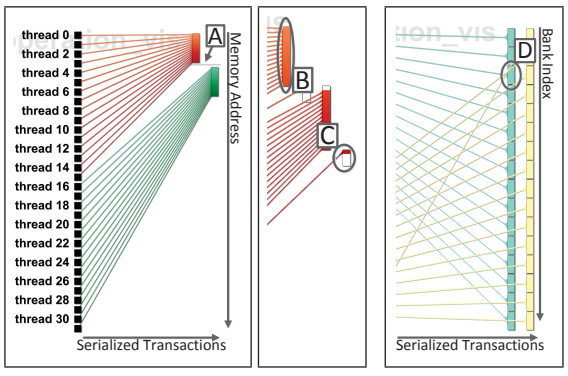
At all levels of the visualization, a source code panel (not shown) is available to the user. As individual or groups of instructions are selected, the source code is highlighted to indicate the connection between the instruction and the issuing lines of source code.

6. Results

We have tested our approach on a number of different algorithms, many with multiple implementations. These include reduction, matrix multiply, matrix transpose, scalar product, vector addition, histogramming, and Haar Wavelet transform. The images in Figures 3, 2(a), 2(b), 4, 5, 7, 8, and 9 have all been generated with our software, though some have been annotated to assist with the description of our approach. We now focus on well understood algorithms, matrix transpose and Haar Wavelet, as case studies for typical behaviors identifiable using our approach. In addition to these case studies, we are actively pursuing optimizations on larger, more sophisticated software.

6.1. Matrix Transpose

A matrix transpose is a fairly simple operation, however, naive implementation can have seriously negative effects on performance. As part of the NVIDIA C/C++ CUDA Samples [NVI12a], three versions of the matrix transpose kernel are provided, each with very different performance results. Our approach makes quite obvious the reason for this variation. Our trace performed the transpose on a 512x512 matrix using 1024 (32x32) blocks of 256 threads (8 warps) each for a total of over 260k threads across the entire grid. Each thread is responsible for a single matrix component, and depending upon the kernel version, the execution for each thread/warp takes between 30 and 50 instructions to complete.



(a) Global memory (b) Shared memory

Figure 5: *Breakdown of operation visualization where: A represents a discontinuity of address space; B shows a single coalesced transaction; C shows a underutilized transaction; and D shows a single bank transaction.*

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

6.1.1. Identifying Good Memory Access Patterns

The naive implementation of matrix transpose has each thread simply read an element from the input matrix, stored in global memory, and directly write those values back out to their location in the output matrix, also located in global memory (Listing 1). Figure 3(a) is an example of behavior for this type of kernel. The problem with this approach is that while the reading of the data is coalesced and quite efficient (Figure 3(d)), the writing out of the data has a large stride in address (i.e. a scatter operation) and produces many non-coalesced writes (Figure 3(e)). These operations flood the memory controller by issuing 32 transactions per warp, and they flood the memory bus since each 4 byte component requires a 32 byte transaction, causing each warp to move 1024 bytes instead of the 128 bytes of actual data.

Listing 1: *Naive Matrix Transpose*

```
1 Transpose(flt *out, flt* in, int w, int h){
2   // Index to data element
3   int idX = blkId.x * TILE_D + thrdId.x;
4   int idY = blkId.y * TILE_D + thrdId.y;
5
6   // Index into input and output arrays
7   int idIn  = idX + w * idY;
8   int idOut = idY + h * idX;
9
10  // Transpose element
11  for(int i=0; i<TILE_D; i+=BLK_ROWS)
12    out[idOut+i] = in[idIn+i*w];
13 }
```

To improve upon the naive implementation, shared memory can be used as temporary storage for the input data, enabling "reordering" of the data for writing. This version loads 16x16 submatrices into shared memory and performs the transpose within the shared memory, allowing for coalesced writing to global memory (Listing 2). Figure 3(b) demonstrates this approach in action. The writing of the data is now performed with two coalesced memory transactions. However, the alignment of addresses used in the shared memory is problematic because the 16-way bank conflict lead to 32 serialized transactions (Figure 3(f)).[§] Figure 6(left) shows the bank layout of the submatrices used in the shared memory read and write. Although this is an undesirable behavior, the cost of using shared memory is still significantly lower than the cost of using non-coalesced global memory.

Listing 2: *Shared Memory Matrix Transpose*

```
1 Transpose(flt *out, flt *in, int w, int h){
2   // Shared memory for temporary storage
3   __shared__ flt tile[TILE_D][TILE_D];
4
5   // Index into input array
6   int idX = blkId.x * TILE_D + thrdId.x;
7   int idY = blkId.y * TILE_D + thrdId.y;
8   int id  = idX + idY * w;
9
10  // Read element from input to shared
11  for(i=0; i<TILE_D; i+=BLK_ROWS, id+=w)
12    tile[thrdId.y+i][thrdId.x] = in[id];
13
14  // Ensure shared is synchronized
15  __syncthreads();
16
17  // Index into output array
18  idX = blkId.y * TILE_D + thrdId.x;
19  idY = blkId.x * TILE_D + thrdId.y;
20  id  = idX + idY * h;
21
22  // Write element from shared to output
23  for(i=0; i<TILE_D; i+=BLK_ROWS, id+=h)
24    out[id] = tile[thrdId.x][thrdId.y+i];
25 }
```

A final implementation improves upon both of the prior approaches. The stride of access within the shared memory is the source of all of the shared memory conflicts. Submatrix blocks are 16 elements wide causing all threads to access the same shared memory bank. Padding the matrix storage [rIC12] has been shown to improve caching performance in these situations by changing the effective stride of memory access. Figure 3(c) shows this approach in practice. Although our submatrix blocks are still 16 elements wide, the shared memory used for this operation is 17 blocks wide, causing subsequent rows to begin at different shared memory banks (Listing 3 and Figure 3(g)). Figure 6(right) shows the new bank layout of the submatrices used in the shared memory read and write. The padding of the matrix has corrected the striding issue formerly present. This final approach solves both the global memory coalescence and shared memory conflict problems simultaneously.

Listing 3: *Optimal Matrix Transpose*

```
1 Transpose(flt *out, flt *in, int w, int h){
2   // Pad store for conflict-free access
3   __shared__ flt tile[TILE_D][TILE_D+1];
4                     . . .
```

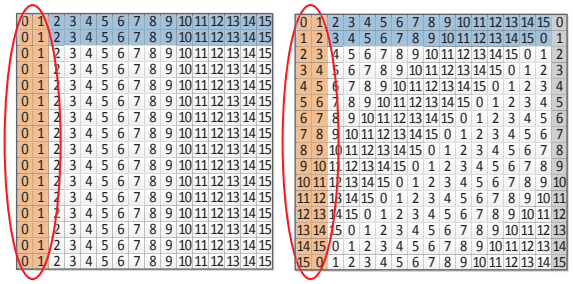


Figure 6: *Diagram of banks used in shared memory matrix transpose. Left: The* 16x16 *matrix sub-blocks allow writing of data (blue) to be conflict-free, while reading data produces conflicts (orange). Right: Padding each row with an unused element results in conflict-free reads and writes.*

[§] In CUDA 1.x, each half-warp is serviced separately for shared memory transaction. Therefore, each half-warp suffers from a 16-way bank conflict for 32 serialized transactions. Changes in CUDA 2.x cause an 8-way conflict with 8 serialized transactions.

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

To further verify the results, the performance of these three implementations were tested on real hardware. These experiments where performed using an NVIDIA GeForce GTX 580. For the naive version of the matrix transpose, the kernel took 0.072 second to complete. For the second version with coalesced global access and shared memory conflicts, the kernel took only 0.040 seconds, a 44% improvement over the naive version. The final, coalesced and conflict free version took 0.027 seconds, a 32% and 62% improvement over the previous and naive versions, respectively.

We have also run these three algorithms through NVIDIA's Visual Profiler as seen in Figure 1. The area outlined in red on Figure 1(a) shows that the profiler identifies the high overhead cost of global and shared memory bottlenecks. Each column of Figure 1(b) shows additional output from the naive, shared memory, and optimized versions of transpose, respectively. The areas outlined in red indicate shared memory, global store, and global load performance, respectively. Finally, not pictured is the analysis tool which identifies the exact line of code producing global memory overhead. These tools are effective at identifying the existence of the same performance bottlenecks as identified in this section. However, the details of *why* are left unanswered.

6.2. Haar Wavelet Transform

A Haar Wavelet is a wavelet formed by a linear combination of square-shape functions. The source code used to generate this trace is the 1D Haar Wavelet example also provided in the NVIDIA C/C++ CUDA Samples [NVI12a]. Our trace performed the transform on a dataset of over 260K elements using 16 blocks of 512 threads (16 warps) each for a total of 8192 threads across the entire grid. The longest executing block performed approximately 550 operations for each thread/warp.

6.2.1. Grid-level and Block-level Variation

Figure 7(a) shows the grid-level view of the Haar Wavelet kernel execution. The first notable feature of this visualization is the large swaths of gray indicating earlier completion of many blocks. This signals that block #1's execution is about 20% longer than any other block. This is a fairly common occurrence among kernels where the first block is used to finalize computation, write output, etc. The other notable feature is that many blocks contain small deviations from the execution of block 1 (see the yellow-orange colored tiles). These deviations are infrequent and probably not worth any further investigation.

Figure 7(b) digs deeper into the execution by visualizing an individual block. The first warp of the block is selected, and the obvious difference is highlighted among all other warps. All warps begin similarly, however that condition quickly changes as execution proceeds. Taking a step further to the warp-level visualization in Figure 7(c) shows a comparison between warp #1 and warp #9 of the block. The difference in execution is apparent as warp #1 executes many instructions while warp #9 remains mostly idle.

6.2.2. Architecture Comparisons

The design of the hardware architecture has an obvious impact on the execution performance of various operations. We have added the capability to switch between CUDA 1.x-3.x architectures for quick comparison of performance. This is accomplished by retaining the original memory reference trace and only simulating the result of memory transactions under a different architecture.

One important change in architecture that appeared between CUDA 1.0 and 1.2 was an improvement to the memory coalescing algorithm used in the hardware. These improvements lead to issuing significantly fewer memory transactions with better utilization. Figure 8 demonstrates this difference with CUDA 1.0 on the top and CUDA 1.2 on the bottom. On the left, with a warp visualization, it is quickly apparent that in the latter portion of the execution, the CUDA 1.0 version issues many memory transactions with poor utilization. Switching to the CUDA 1.2 version, it can be seen that the number of memory transactions issued and overall utilization are improved. The operation-level visualization on the right gives a more detailed view of the problem. For CUDA 1.0, despite the memory access pattern being localized in nature, the accesses are not coalesced. For CUDA 1.2 however, this problem is corrected and the memory access is coalesced.

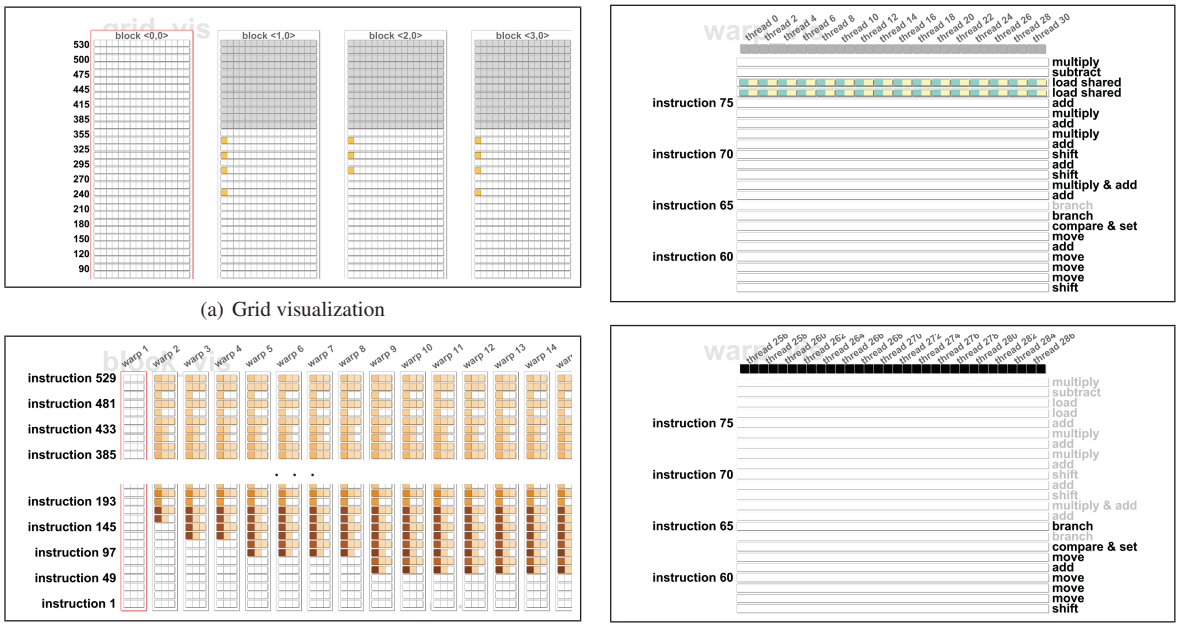
One significant change between the architectures of CUDA 1.x and 2.x is the number of shared memory banks increased from 16 in CUDA 1.x to 32 in CUDA 2.x.[¶] This has an effect of halving the number of serialized transactions in most, but not all cases. Figure 9 shows an example. The same set of operations for both CUDA 1.x (left) and 2.x (right) are shown. For the first operation (Figures 9(c) and 9(e)), in CUDA 1.x the set of addresses used causes a total of four serialized transactions. With twice as many banks, the CUDA 2.x example only needs two serializes accesses to service the conflict. However, in rare cases the increased bank count does not reduce conflicts. In the second operation (Figures 9(d) and 9(f)), in CUDA 1.x two serialized accesses are needed. However, in CUDA 2.x a 2-way bank conflict persists due to the unusual memory access pattern present.

7. Conclusions

In conclusion, we have presented a novel visual approach to investigating the memory behavior of CUDA kernels. Our approach focuses on identifying representative behaviors at high-levels, followed by detailed investigations at low-levels. We have also demonstrated the approach for multiple kernel implementations with our results confirmed using real hardware. The largest limitation to our approach is its incompleteness, mostly due to trade secrets. We have done our best to conform to the public information regarding the hardware, but many assumptions have been made and some features ignored. In the future, we plan to extend this work to feature additional hardware components and provide tool for the logic of optimization, for example, visualizing memory access in the context of data structure memory.

[¶] In addition, CUDA 2.x does not split into half-warps for servicing.

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

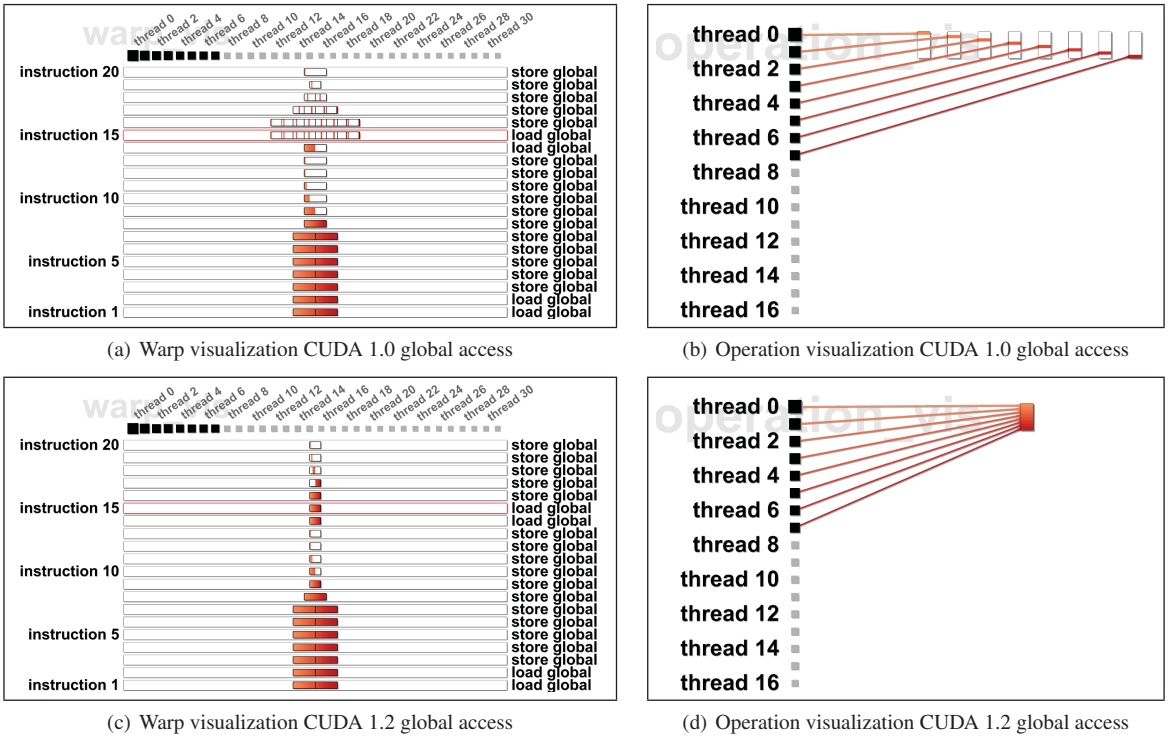


(a) Grid visualization

(b) Block visualization

(c) Warp visualizations of two divergent warps

Figure 7: *Comparative analysis shows little difference at the grid-level (a) but large difference at the block-level (b). Investigation of two warps (c) shows that the difference is that one warp remains active (top) while the other becomes inactive (bottom).*



(a) Warp visualization CUDA 1.0 global access

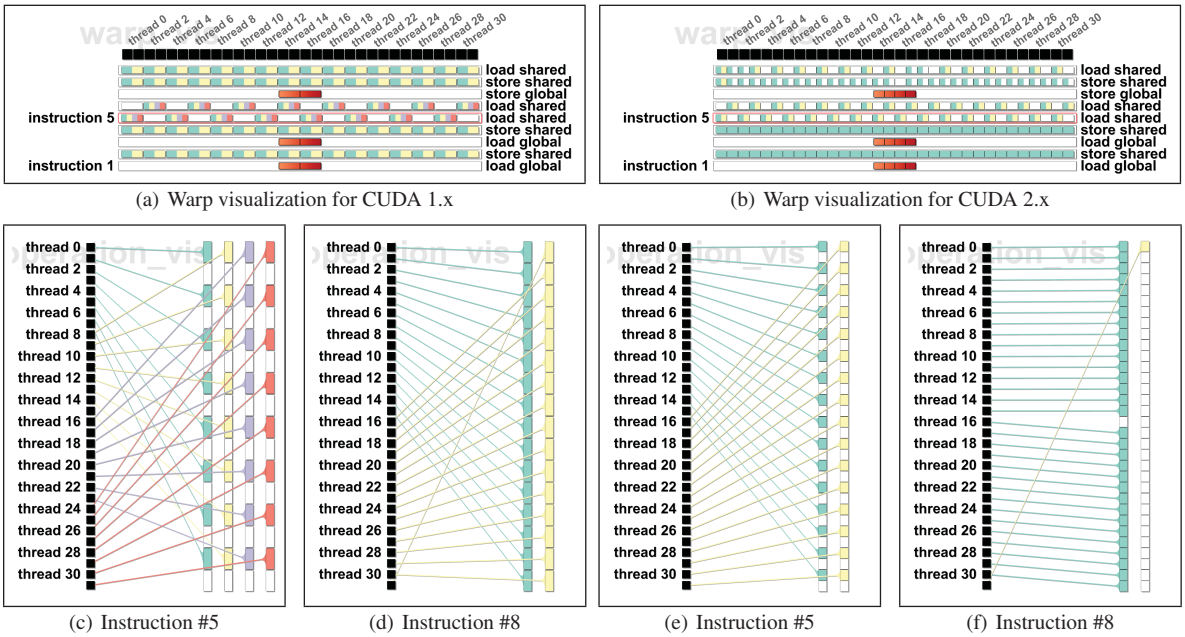
(b) Operation visualization CUDA 1.0 global access

(c) Warp visualization CUDA 1.2 global access

(d) Operation visualization CUDA 1.2 global access

Figure 8: *A comparison of global memory access on different architectures shows that while the majority of memory transactions are similarly coalesced, some transactions will perform significantly better on CUDA 1.2 (d) than on CUDA 1.0 (b).*

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.



(a) Warp visualization for CUDA 1.x

(b) Warp visualization for CUDA 2.x

(c) Instruction #5

(d) Instruction #8

(e) Instruction #5

(f) Instruction #8

Figure 9: *A comparison of a shared memory access on different architectures shows that some instructions ((c) and (e)) improve as expected while others do not ((d) and (f)).*

Acknowledgments

We wish to thank Kristi Potter for her valuable feedback. This research was supported in part by grants from DOE NETL and King Abdullah University of Science and Technology (KAUST) award ID KUS-C1-016-04.

References

[App12] APPLE INC.: Instruments user guide, 2012. 2

[BBH08] BERNARDIN T., BUDGE B. C., HAMANN B.: Stacked-widget visualization of scheduling-based algorithms. In *4th ACM symposium on Software visualization* (2008), SoftVis '08, pp. 165–174. 2

[DKYC10] DIAMOS G., KERR A., YALAMANCHILI S., CLARK N.: Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Parallel architectures and compilation techniques* (2010), pp. 353–364. 2

[GNU12] GNU: GNU gprof, 2012. 2

[Int12] INTEL CORP.: Intel VTune Amplifier XE 2013, 2012. 2

[KDY11] KERR A., DIAMOS G., YALAMANCHILI S.: Gpu application development, debugging, and performance tuning with gpu ocelot. *GPU Computing GEMS 1* (2011). 2

[MT07] MORETA S., TELEA A.: Visualizing dynamic memory allocations. In *IEEE Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)* (2007), pp. 31 –38. 2

[NAW*96] NAGEL W., ARNOLD A., WEBER M., HOPPE H., SOLCHENBACH K.: VAMPIR: Visualization and analysis of MPI resources. *Supercomputer 12*, 1 (1996), 69–89. 2

[NS03] NETHERCOTE N., SEWARD J.: Valgrind: A program supervision framework. In *Workshop on Runtime Verification* (2003). 1

[NVI12a] NVIDIA: CUDA CODE SAMPLES, 2012. 6, 8

[NVI12b] NVIDIA: *CUDA Programming Guide 4.2*. 2012. 2, 3

[NVI12c] NVIDIA: Nsight. `http://www.nvidia.com/object/nsight.html`, 2012. 1, 2

[NVI12d] NVIDIA: Visual profiler, 2012. 2

[rIC12] A.N.M IMROZ CHOUDHURY: *Visualizing Program Memory Behavior Using Memory Reference Traces*. PhD thesis, University of Utah, Salt Lake City, UT, 2012. 7

[rICPP08] A.N.M IMROZ CHOUDHURY, POTTER K., PARKER S.: Interactive visualization for memory reference traces. *Computer Graphics Forum 27*, 3 (2008), 815–822. 2

[rICR11] A.N.M IMROZ CHOUDHURY, ROSEN P.: Abstract visualization of runtime memory behavior. In *IEEE Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)* (2011). 2

[SLB*11] SCHULZ M., LEVINE J. A., BREMER P.-T., GAMBLIN T., PASCUCCI V.: Interpreting performance data across intuitive domains. In *Parallel Processing* (2011), pp. 206–215. 2

[SM06] SHENDE S., MALONY A.: The tau parallel performance system. *J. of High Performance Computing Applications 20* (2006), 287–311. 2

[TJYD09] TERPSTRA D., JAGODE H., YOU. H., DONGARRA J.: Collecting performance data with PAPI-C. In *Tools for High Performance Computing* (2009), pp. 157–173. 1

[Tot10] TOTALVIEW TECHNOLOGIES: Case studies. `http://www.totalviewtech.com/support/case_studies.html?via=resources`, 2010. 1

[Uni12] UNIVERSITY OF OREGON: Tau - tuning and analysis utilities, 2012. 2

[Zen12] ZENTRUM FÜR INFORMATIONSDIENSTE UND HOCHLEISTUNGSRECHNEN: Vampirtrace, 2012. 2

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.