Memory Access Analysis for CUDA programs

Prithayan Barua
School of Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: prithayan@gatech.edu

***Abstract*—CUDA is a very popular and easy to use programming model for NVIDIA GPUs. But now it is being targeted at all kinds of parallel architectures. CUDA has an interesting programming paradigm that exposes new opportunities for compiler analysis and optimization aimed at new applications. In this project we analyze the memory access pattern of CUDA kernels. We present an LLVM compiler pass that can be used to analyze all the memory accesses made inside a kernel. The pass can instrument the kernel to print the block stride information for every memory access. We explore a static compile time version and another dynamic runtime version for the pass. We have used profiled block strides to verify our compiler pass. Block stride information can be utilized for various different optimizations, like deciding the interleaving granularity of memory layout for PIM systems, or for improving bandwidth utilization in GPU systems and also for improving scheduling in CPU multicore systems.**

I. Introduction

CUDA is a very popular general purpose parallel programming model that is targeted at the highly parallel NVIDIA GPUs. CUDA has extended the C programming model with functions called kernels. The programmer defines the work to be done by a single thread using a kernel and then N different CUDA threads are launched in parallel, all of which execute the same kernel. CUDA threads have multiple levels of hierarchy, a Thread Block is a group of threads that can synchronize and share data and they run on the same Streaming Multiprocessor. Thread blocks are then grouped into Grids of Threads. The programmer can control the dimensionality of the thread blocks and grids. Each thread can uniquely identify itself using the thread id within a thread block and the thread block id within the grid. Figure 1 from the CUDA programming guide[1] shows the concept of grid of thread blocks.
Although CUDA was designed specifically for NVIDIA GPUs, it has become quite popular and is now being used for targeted at various different architectures including CPUs and FPGAs.

A. PIM systems

The motivation for this project was managing data layout for Processing in Memory(PIM) systems. A PIM system is composed of several PIM stacks that serve as the main memory and a host processor. In a PIM stack, compute logic and memory cells are tightly coupled, which means the processor logic is directly connected to the memory on the stack. So if an instruction running on a PIM stack is accessing the memory on the same stack, it can utilize the high bandwidth memory transfer effectively. But if the data being accessed is on another PIM stack, it has to rely on a low bandwidth channel to other PIM cores.
The memory mapping scheme can control the granularity to either fine-grain-interleaved regions (FGR) or coarse-grain-interleaved regions (CGR). FGR policy maps a single page across multiple consecutive PIM stacks while CGR maps an entire page contiguously on one stack. FGR helps to maximize memory bandwidth by distributing concurrent accesses across all available memory interfaces. But CGR would be more beneficial if an entire memory object is co-located with the program that accesses it.
GPU programming model can be used to program such PIM systems. The host processor can launch threads corresponding to the GPU kernels on multiple PIM stacks. Each PIM stack can be treated like a Streaming Multiprocessor, and one thread block is run on a single PIM stack.
Now the objective is given the CUDA program how to decide between FGR and CGR layout. We consider this problem in detail in the following sections.

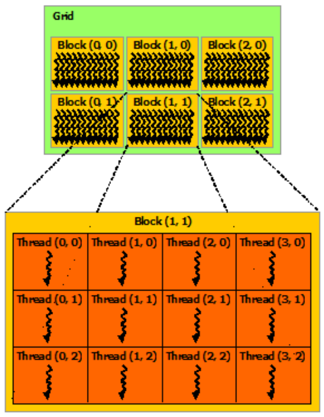


Fig. 1: CUDA Grid of Thread Blocks

II. Thread Block Stride

Given a GPU kernel we know that each thread block will execute on a single PIM stack. The basic motivation behind the memory layout is co-locating the data and the instructions that operates on it. If we can find out that each thread block is accessing an exclusive set of memory locations, then CGR should be more beneficial. While if we know that the thread blocks mostly share data and they access the same set of addresses then FGR which distributes the shared data evenly should be beneficial.

Now we narrow down our problem to the individual array accesses within a kernel. Given the index expression for each memory access, our objective is to decide if it results in a shared or exclusive access pattern. We want to design a static compiler pass that can analyze each array access and decide the memory layout depending on the access pattern.

If a thread block accesses the first $block\_stride$ bytes of a memory object and then each consecutive thread block accesses the next $block\_stride$ bytes, then we have an exclusive access pattern with a fixed block stride. So if we can compute this block stride before launching the GPU kernel, then the memory allocation unit can do a Coarse Grained layout of the object with $block\_stride$ bytes on each PIM stack. While if a non zero block stride does not exist then an FGR policy can be adopted.

Next lets look at the index expressions of arrays to understand the access pattern.

A. Example

Consider the example in Figure 2, the left frame shows the CUDA code sample and the table shows the array location accessed by each thread. the first column and row is the block index, the second column and row show the thread indices. The table has been color coded to identify the threads within the same block. As we can see this access pattern has a nice block stride. Each thread block accesses the 4 contiguous locations of the array, and it wraps around the Y dimension to evenly distribute every memory location to one single thread.

B. Index Computation

Lets first consider, what are the possible variables that can be used in an array index expression inside a GPU kernel. Because of the CUDA programming model, any index expression can be expressed in terms of

- Kernel function parameters and the thread block and grid dimensions ($blockDim$).
- The Block Index($blockIdx$) or Thread Index($threadIdx$)
- Memory loads.

We do not include kernel local variables, because they can always be expressed in terms of the above variables, and even loops in the kernel can be virtually unrolled to get rid of the loop indices. Lets consider the above listed variables in detail. Kernel function parameters are the arguments passed by the host code. What is interesting is that every thread receives the same set of function parameter values, so basically the function parameters are constant across threads of the same kernel invocation. Even the thread block dimensions (which are a kind of kernel parameters) are constant as far as the kernel is concerned. The thread index and the block index obviously help to identify each thread uniquely, and the interesting observation here is that they have a fixed pattern, and are predictable. But the last kind of variables, the memory loads, are unpredictable and can vary across thread blocks in an irregular manner. In this project we implemented two versions of block stride computation pass, one is the static version that cannot handle memory loads and another runtime version, that can use memory values to compute a block stride. For now lets ignore the memory loads and consider that we cant handle the memory values, in later sections we will see how to deal with memory loads. So for now if any index expression depends on memory loads, we cannot analyze that memory access. So the only variables in our index expression are the block index and thread index, since the kernel parameters are constants and we ignore accesses with memory loads.

Hence the array index can be expressed as a function of the block index and thread index.

$$index = f(Block\_Index, Thread\_Index) \tag{1}$$

Now for a particular block index B and thread index T, the block stride can be defined as

$$block\_stride = f(B+1, I) - f(B, I) \tag{2}$$

If we have a constant block stride, that means it does not depend on the thread index or block index. So if an array index expression has a constant block stride, then it can be expressed only in terms of the kernel parameters.

$$block\_stride = g(kenel\_parameters) \tag{3}$$

That is if there exists a constant block stride that can be expressed in terms of the kernel parameters, then it can be computed before the kernel is launched by the host. And can be used to decide the memory layout of any memory object before the kernel invocation.

C. Example

Lets consider an array A and one of the most common index expressions in CUDA,

$$A[blockIdx.x * blockDim.x + threadIdx.x] \tag{4}$$

then the block stride can be computed as,

$$\begin{aligned} block\_stride =& ((B+1) * blockDim.x + I)) \\ & - (B * blockDim.x + I)) \qquad (5) \\ \implies block\_stride =& blockDim.x \end{aligned}$$

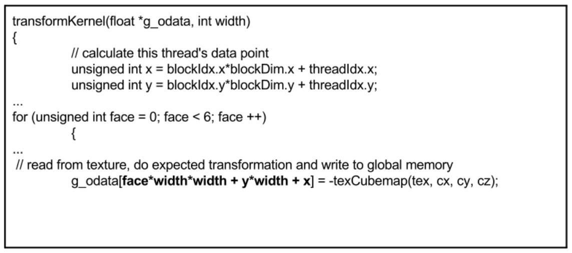
Our experiments have shown the $blockDim$, or some multiple of it is actually the most common block stride.

In the next sections we discuss how to design a compiler pass that can analyze the index expression to deduce the block stride expression.

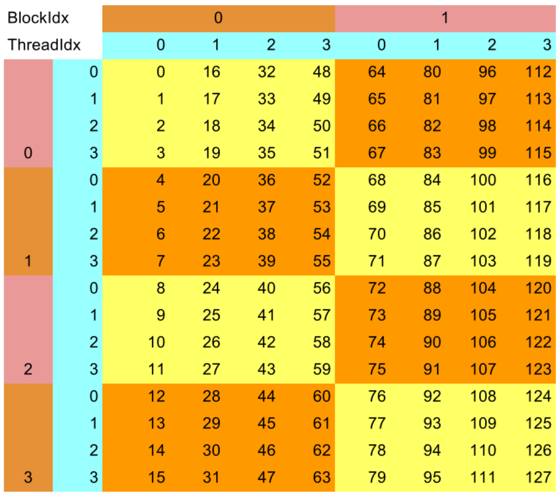
III. Mechanism

We use the LLVM compiler framework to implement the block stride computation pass. The LLVM compiler has recently[2] supported compilation of CUDA framework.

There are still a few open bugs like the Bug ID 26400, because of which few common benchmarks cannot be compiled by clang. We use a combination of clang and nvcc to instrument the CUDA source code and generate an executable.



```
transformKernel(float *g_odata, int width)
{
            // calculate this thread's data point
            unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
            unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
...
for (unsigned int face = 0; face < 6; face ++)
            {
...
 // read from texture, do expected transformation and write to global memory
            g_odata[face*width*width + y*width + x] = -texCubemap(tex, cx, cy, cz);
```



BlockIdx		0				1			
ThreadIdx		0	1	2	3	0	1	2	3
	0	0	16	32	48	64	80	96	112
	1	1	17	33	49	65	81	97	113
	2	2	18	34	50	66	82	98	114
0	3	3	19	35	51	67	83	99	115
	0	4	20	36	52	68	84	100	116
	1	5	21	37	53	69	85	101	117
	2	6	22	38	54	70	86	102	118
1	3	7	23	39	55	71	87	103	119
	0	8	24	40	56	72	88	104	120
	1	9	25	41	57	73	89	105	121
	2	10	26	42	58	74	90	106	122
2	3	11	27	43	59	75	91	107	123
	0	12	28	44	60	76	92	108	124
	1	13	29	45	61	77	93	109	125
	2	14	30	46	62	78	94	110	126
3	3	15	31	47	63	79	95	111	127

Fig. 2: An example access pattern

Our pass takes as input the LLVM IR representation of CUDA, then we iterate over all the instructions and filter out the array indexing instructions for further processing. The array index instructions are called $GetElementPtrInst$ instruction in LLVM. Then we use the def-use graph to trace the definition of all the sources of the $GetElementPtrInst$. If we can trace it back to the $blockIdx$ that is a call to $llvm.nvvm.read.ptx.sreg.ctaid.x$ that means the index can be expressed in terms of the block index, otherwise the index does not depend on the block index. If the array index instruction does not depend on the block index, that means every block accesses the same set of memory locations, and thus we have a shared access pattern, which means the block stride is zero. Lets look at the basic idea behind the compiler pass,

- Get the $GetElementPtrInst$ index instruction.
- Recursively trace the source operands to their root definition until $blockIdx$ is found.
  - if memory load is found then skip the index, block stride cannot be computed.
  - if $blockIdx$ is not found, then block stride is zero, finish computation.
  - else continue to the next step.
- Clone the $GetElementPtrInst$ instruction to $index_k$, and replace $blockIdx$ with a constant K.
- Create another clone of the $GetElementPtrInst$ instruction $index_{k-1}$, and now replace $blockIdx$ with a constant $K-1$.
- Create a Subtract instruction $block\_stride = index_k - index_{k-1}$, and perform algebraic simplification on it.
- if after simplification the $block\_stride$ contains only the kernel parameters and the block and thread indices are canceled out
  - Then we have a const block stride, instrument the CUDA code to insert the $block\_stride$ instruction and print at runtime.
  - else The access pattern does not have a constant block stride.

With the basic idea as listed above, we actually need not perform all the steps. We can take an index instruction and analyze its source operands and operators to symbolically perform the subtract operation. Firstly we replace all the operators in the index expression with equivalent basic algebraic operators, that is $*, /, +, -$. This mainly involves handling bitwise operators. and identifying patterns in them. After we have a mathematical expression, the only terms that are important in this expression are the ones that are multiplied or divided to the $blockIdx$. Because others will cancel out when performing the actual subtraction $index_k - index_{k-1}$. If we are able to perform this symbolic subtraction and get an expression for block stride, then we instrument the code to print the block stride, else we print that the block stride cannot be computed.

A. Handling Control Flow

PHI nodes are used by the LLVM IR to represent an SSA form, such that every use has exactly one reaching definition. A PHI node selects one of the incoming values depending on the particular control flow path that lead to that PHI node. PHI nodes can only occur at the beginning of basic blocks. So the PHI nodes express the control flow structure within the IR. For our objective we have to decide which of the incoming values to a PHI node should we consider for computing the block stride. The basic idea is that we consider only the initial value that reaches a PHI node. So among the incoming values, we consider the Instruction that dominates the PHI node. For this purpose we use the "DominatorTreeWrapperPass"analysis of LLVM. For any given PHI node we ignore all the incoming values which do not dominate the PHI node.

B. Runtime Stride Computation

In our first implementation we did not handle memory operations, and hence could not compute the stride for array expressions which involve memory loads. Furthermore in the first version, if we could not algebraically simplify the block stride expression to constants, during compile time, then block stride could not be computed.
We did another implementation where we wanted to handle all these cases. Now, instead of deciding if the block stride exists at compile time, we can postpone the decision to runtime, before the kernel is launched. So now we instrument the code to compute the particular strides for 3-4 random blocks. And then just insert an if statement to compare if they are all same. If the stride matches then we have a constant block stride, else we donot.
We take advantage of the fact that an accurate stride analysis is not necessary, since it is just used to decide the memory layout and does not impact correctness. So if at runtime we find out that the stride is identical for at least a 2-3 different consecutive blocks, then we can consider that we have a const block stride. The overhead of this method is just a few extra instructions.

C. Memory Loads

Postponing our block stride computation to runtime will let us handle the memory loads to some extent. Firstly we have to instrument the host code to compute the block stride before the kernel is launched. So, the only memory values that are possible to be handled are the memory values that are not written to inside the kernel.
So we have to scan the kernel and record all the array variables that are being written to inside a kernel. If the index expression loads values from the memory which belongs to the write set of the kernel, then it is not possible to use this value before launching the kernel. Hence we cannot compute the block stride for array expressions which depend on the memory values written to inside the kernel.
But if the memory is read-only inside the kernel, then we can use such memory loads to compute the block stride.

IV. Evaluation Methodology

We have used several GPU benchmarks including the graph-big, parboil, rodinia and CUDA sdk to test our compiler pass. We are using the clang compiler framework to call our pass to instrument the source code for block stride computation. Then nvcc compiler is being used to link them to the executable. Ideally we should instrument the host code to compute the block stride, but the CUDA compilation sequence is not straightforward. They use gcc to compile the host code, then clang/NVCC to compile the device code, and then NVCC to create the executable. So instead of attempting to write multiple passes and hack this sequence, we work only on the kernel code. We have instrumented the kernel code with printf statements to print the block stride at runtime. We also need to make sure that it prints the information only once, otherwise every CUDA thread will overflow the output with prints. For this purpose we have added a wrapper function and guarded it with block id and thread id, to print it only once in one particular thread.
We have manually verified the block strides as computed by our pass with the source code, and also with some profiled block strides that were available.

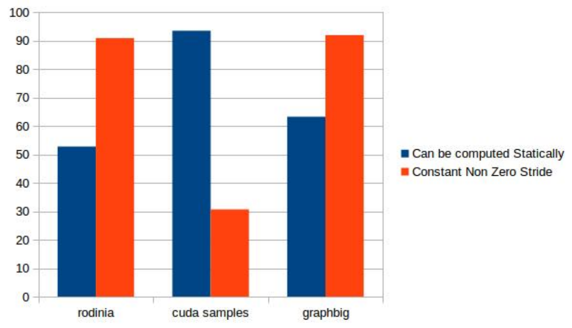


Fig. 3: Results of Static Block Stride Pass

A. Profiled Block Strides

The profiled block strides had the PC address and the block stride corresponding to that. But since our compiler generated stride did not have a PC address, we had an issue in mapping the profiled strides to line numbers. We found the NVIDIA tools, cuobjdump and nvdisasm, that can be used to map the binary and PTX to source line numbers. We planned to use this tool to get the PC address to line number mapping. But we found out, in most cases there were 3 to 4 block strides computed by our pass, so it was easy to verify just the numbers with the profiled output. Manually mapping the PC to line number was quite time consuming so we skipped it.

V. Results

In our project the results are just the block stride numbers. But we found it interesting to plot the kind of pattern in block strides in various benchmarks. We ran our static implementation, that prints the block stride expression at compile time on a few benchmarks. Then we computed the percentage of memory accesses that can be handled by our pass, and the percentage of memory accesses that have a constant non zero block stride. The figure 3 shows the results of this analysis. For example in the CUDA samples, we analyzed 2931 memory accesses out of which 190 accesses could not be handled by our static pass.
This data can help visualizing interesting patterns in the benchmarks. We can see that the cuda samples has mostly implemented simple access patterns that can be predicted statically, and they have a shared access pattern for about 70% of the accesses.

VI. Discussion

After implementing both the static and runtime version of our pass, we made some interesting observations.
Even if we handle the read-only memory loads, to compute block stride, it does not help much. The only case when it can

be used is if every block stride loads the same memory values. That is the memory load does not depend on the block index. So if we know that every thread block is going to load the same memory value, only then we can pre-load that memory address and compute the block stride before kernel is launched. So memory loads that may depend on thread-index but not on block index can be used to compute the block stride.
In all other cases, until we see all the accesses made by all the threads, it may not be possible to compute the stride. That is if the block stride depends on some particular array, until we iterate over the entire array it maynot be possible to predict the block stride.
Other than the memory loads, some control flow also makes it difficult to compute the block stride. If a memory access is heavily nested inside if else, or for loop structures, then it is not possible to resolve the exact array index expression. But thankfully most CUDA codes donot have too many control flow structures.
I hope the block stride computation will be an essential part of the LLVM compiler and should be available as a library call, just like the loop index analysis for sequential code. In the next section we will see why block stride is so important and relevant to many different users.

VII. Related Work

Even though the block stride computation was motivated by the memory layout decision for PIM systems, the block stride is a very important analysis information that can be used for other systems.
The block stride provides a very important information regarding the sharing of memory addresses between adjacent thread blocks. Accesses to global memory are a major performance bottleneck in GPU systems. So if the compiler can detect that two adjacent thread blocks are accessing the same set of memory locations, then there is a possibility of merging the thread blocks [3]. The compiler can use certain heuristics to modify the thread block dimensions to merge two adjacent blocks into one. Now, the common set of memory locations can be loaded once, and shared using shared memory or registers within a single Streaming Multiprocessor.
The compiler can use the block stride information to detect partition camping[4]. Since consecutive thread blocks are likely to be placed on adjacent SMs, they are likely to be active at the same time, and accessing the same set of instructions concurrently. Partition camping occurs when the accesses from multiple SMs queue up on the same memory controller when accessing the global memory. So given the block stride, we can detect partition camping if the stride is a multiple of $(partition\_size * number\_of\_partitions)$, that is two consecutive thread blocks concurrently access the same partition. Then the compiler can even perform some transformations on the access pattern to avoid partition camping [5].
We can also use the block stride information for designing a hardware/software prefetching[6] mechanism for the GPU. The hardware can use the block stride information passed on by the compiler to exactly compute the addresses that will be accessed by the consecutive thread blocks, and prefetch them before they are requested.
The block stride has also been used for scheduling work-items from existing OpenCL to CPUs [7]. They use the block stride information to perform data-locality centric work-item scheduling.

VIII. Future Work

The future work mainly has two directions. Firstly we can try to improve the coverage of our block stride pass. We have to experiment further and handle cases that our pass does not handle correctly. We can also improve the way memory loads are handled.
The second direction is to use this block stride information to implement some compiler optimization passes. We can implement certain analysis passes and enable compile time transformations to improve the memory layout for GPUs or improve memory bandwidth, as suggested in Related Works section.

IX. Conclusion

In this project we mainly explored the analysis of array index expressions in CUDA kernels. We developed our pass on the LLVM framework for CUDA. The CUDA programs have a nice predictable pattern of memory accesses because of the use of nice predictable thread and block strides. Unlike the sequential loop indices, where the analysis is much more complicated and not always feasible. We were able to instrument the CUDA kernel with instructions to compute and print the block stride information. Then depending on the kind of applications, we were able to compute the block stride accurately. We also saw how the block stride can reveal interesting information about the applications and can be used to facilitate different kinds of optimizations for different architectures.

References

[1] NVIDIA, “Cuda toolkit documentation.” [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[2] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, “Gpucc: An open-source gpgpu compiler,” in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO ’16. New York, NY, USA: ACM, 2016, pp. 105–116. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854041

[3] A. Magni, C. Dubach, and M. O’Boyle, “Automatic optimization of thread-coarsening for graphics processors,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT ’14. New York, NY, USA: ACM, 2014, pp. 455–466. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628087

[4] A. M. Aji, M. Daga, and W.-c. Feng, “Bounding the effect of partition camping in gpu kernels,” in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF ’11. New York, NY, USA: ACM, 2011, pp. 27:1–27:10. [Online]. Available: http://doi.acm.org/10.1145/2016604.2016637

[5] Y. Yang, P. Xiang, J. Kong, and H. Zhou, “A gpgpu compiler for memory optimization and parallelism management,” *SIGPLAN Not.*, vol. 45, no. 6, pp. 86–97, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1809028.1806606

[6] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, “Guided region prefetching: A cooperative hardware/software approach,” in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ser. ISCA ’03. New York, NY, USA: ACM, 2003, pp. 388–398. [Online]. Available: http://doi.acm.org/10.1145/859618.859663

[7] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu, "Locality-centric thread scheduling for bulk-synchronous programming models on cpu architectures," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 257–268. [Online]. Available: http://dl.acm.org/citation.cfm?id=2738600.2738632