15754 Project

Salil Joshi & Cyrus Omar

April 27, 2011

1 Introduction

Modern graphics processing units (GPUs) have for the first time made high performance parallel computing affordable and widely available. GPUs are being rapidly adopted for use in applications that would otherwise require large computing clusters and an increasing number of programmers find themselves writing code for GPUs. However, GPU program optimization is currently done largely by hand. Optimizing compilers for GPUs are a fairly recent development and the literature on formal methods for performing the analyses required for GPU program optimization is sparse.

In this project, we describe a formal system for analyzing GPGPU programs in order to determine what optimizations are applicable. Our system tracks and analyzes array access patterns in OpenCL programs. In particular, we track array accesses that are linear polynomials of the thread and block indices. This form of array access is extremely common in GPGPU programming, and thus an important potential point of optimization.

We define a simple language (that is essentially a subset of OpenCL) and a type system for this language. The type system has a novel type $\mathbf{ID}(a, b, c)$ for thread and block indices. The process of assigning types to terms in this language builds up contexts that contain a list of array accesses through the use of this $\mathbf{ID}(a, b, c)$ type. We utilize this information in order to determine the array indices and memory segments accessed by a half warp. We can then determine the coalescing behavior of threads in the program, and identify potential data sharing between threads in different blocks or even in different kernels.

If uncoalesced accesses are identified by our system, a program transformation like the one described in [5] can be used to convert these to coalesced accesses, which can provide a significant speedup. Data sharing information can be used to make decisions about merging adjacent blocks or corresponding threads in different blocks or different kernels. Here a speedup is obtained by using the block-local shared memory for shared data segments.

Our main contributions are:

- We have developed a formal system to track array accesses that are linear polynomials of thread and block ids.
- Our system formalizes the analyses used in [5] for intra-kernel optimizations like memory coalescing and thread/block merging.
- We show how our system could be used for *kernel merging*.

1.1 Related Work

Recently there has been a lot of interest in developing optimizing compilers for GPUs [5], [3]. The analysis used in these approaches in order to identify applicable optimizations is quite similar in spirit to ours. Unfortunately these papers only provide informal descriptions of the analyses they use. A major goal of this project was to design a formal system that can adequately capture the analyses required for such optimizations, especially the ones presented in [5]

Similar analyses has also been done in the context of analytical modeling of performance of GPU programs, for example in [2]. Other methodologies for performance modeling such as [1] also require an underlying system to perform these kinds of analyses. However the goals and scope of our analyses are somewhat different. For example, knowing when memory accesses are coalesced is crucial for performance modeling, but identifying *potential* points of data sharing is not. Performance modeling also requires reasoning about other factors (such as register usage) which our system does not deal with.

In general, array access analyses is also done by CPU compilers. The techniques used there may be beneficial for GPUs as well. Our focus however was on those aspects of array accesses which are unique to GPGPU computation: accesses involving thread and block indices, for which there is no direct parallel on CPUs.

2 Analyzing Array Accesses

2.1 Grammar

We will work on a language that is essentially a simple subset of OpenCL.
Types:

$$\begin{aligned} \text{T'} &:= \text{uint} \mid \text{int} \mid \text{float} \mid \text{bool} \\ \text{T} &:= \textbf{global}\ T'[] \mid \textbf{local}\ T'[] \mid T' \mid \textbf{const}\ T' \mid \textbf{ID}(a,b,c) \end{aligned}$$

Terms

$$e := x \mid 1 \ldots \mid tid \mid bid \mid a[e]$$

$$\text{stmt} := \langle\rangle \mid \text{cmd}; \text{stmt}$$

$$\begin{aligned} \text{cmd} := & \textbf{if}\ e\ \textbf{then}\ \text{stmt}\ \textbf{else}\ \text{stmt} \\ & \mid \textbf{for}\ x\ \textbf{in}\ [e,e,e]\{\text{stmt}\} \\ & \mid \textbf{sync} \\ & \mid x := e \\ & \mid a[e] := e \\ & \mid \textbf{decl}\ T\ x\ \textbf{in}\ \text{stmt} \end{aligned}$$

The language is a simple imperative language with a few special features:

- *tid* is special variable that holds the thread index of the current thread. Similarly, *bid* holds the block index.
- Arrays are either **global** or **local**. **global** arrays are stored on the global device memory while **local** arrays are stored on the block level shared memory
- A block level synchronization primitive, **sync** which corresponds to OpenCL's `__syncthreads__`
- **const** types. The value of an expression of this type can be computed at compile time.
- $\textbf{ID}(a,b,c)$. An expression of this type has the value $a * \text{tid} + b * \text{bid} + c$ i.e. it is a linear polynomial in the thread and block indices.

2.2 Typing

We have three typing judgements:

- $\Gamma/\Delta \vdash e : T$ for expressions e states that e has type T in the context Γ. The context Δ collects the set of indices used by e for every global array that e accesses (reads from). However, only those indices which are *linear polynomials in tid and bid* (i.e. have type $\mathbf{ID}(a, b, c)$) are collected. Δ will later be used to determine the memory segments that are accessed by the program, which is useful for optimizations like memory coalescing and data sharing.
- $\Gamma/\Delta_r/\Delta_w \vdash C$ ok for commands C states that the command is well formed and all subexpressions are well typed. In addition, Δ_r is the set of global array indices *read from* by C and Δ_w is the set of global array indices *written to* by C, similar to Δ in the previous judgement. Once again we only track array indices that are linear polynomials in tid and bid.
- $\Gamma/\Delta_r/\Delta_w \vdash S$ ok similar to the previous judgement, but for statements.

2.2.1 Expressions

$$\frac{x : T \in \Gamma}{\Gamma/\cdot \vdash x : T} \qquad \frac{}{\Gamma \vdash \text{tid} : \mathbf{ID}(1,0,0)} \qquad \frac{}{\Gamma \vdash \text{bid} : \mathbf{ID}(0,1,0)}$$

$$\frac{\Gamma/\Delta_1 \vdash e_1 : \mathbf{ID}(a_1, b_1, c_1) \qquad \Gamma/\Delta_2 \vdash e_2 : \mathbf{ID}(a_2, b_2, c_2)}{\Gamma/\Delta_1 \cup \Delta_2 \vdash e_1 + e_2 : \mathbf{ID}(a_1 + a_2, b_1 + b_2, c_1 + c_2)}$$

$$\frac{\Gamma/\Delta_1 \vdash e_1 : \mathbf{ID}(0,0,c') \qquad \Gamma/\Delta_2 \vdash e_2 : \mathbf{ID}(a,b,c)}{\Gamma/\Delta_1 \cup \Delta_2 \vdash e_1 * e_2 : \mathbf{ID}(a*c', b*c', c*c')}$$

$$\frac{\Gamma \vdash a : \mathbf{global}\ T'[] \qquad \Gamma/\Delta \vdash e : \text{uint}}{\Gamma/\Delta \vdash a[e] : T'} \qquad \frac{\Gamma/\Delta \vdash a : \mathbf{global}\ T'[] \qquad \Gamma/\Delta \vdash e : \mathbf{ID}(a', b', c')}{\Gamma/(a, (a', b', c')), \Delta \vdash a[e] : T'}$$

$$\frac{\Gamma/\Delta \vdash e_1 : \text{uint}}{\Gamma/\Delta \vdash e_1 : \text{int}} \qquad \frac{\Gamma/\Delta \vdash e_1 : \mathbf{const}\ \text{uint}}{\Gamma/\Delta \vdash e_1 : \mathbf{ID}(0,0,e_1)} \qquad \frac{\Gamma/\Delta \vdash e_1 : \mathbf{const}\ T'}{\Gamma/\Delta \vdash e_1 : T'}$$

2.2.2 Commands

$$\frac{x : T, \Gamma/\Delta_r/\Delta_w \vdash S\ \text{ok}}{\Gamma/\Delta_r/\Delta_w \vdash \mathbf{decl}\ T\ x\ \mathbf{in}\ S\ \text{ok}} \qquad \frac{x : T \in \Gamma \qquad T\texttt{scalar} \qquad \Gamma/\Delta_r \vdash e : T}{\Gamma/\Delta_r/\cdot \vdash x := e\ \text{ok}} \qquad \frac{}{\Gamma/\cdot/\cdot \vdash \mathbf{sync}\ \text{ok}}$$

$$\frac{\Gamma/\Delta_r^1 \vdash e : \text{bool} \qquad \Gamma/\Delta_r^2/\Delta_w^2 \vdash S_1\ \text{ok} \qquad \Gamma/\Delta_r^3/\Delta_w^3 \vdash S_2\ \text{ok}}{\Gamma/\Delta_r^1 \cup \Delta_r^2 \cup \Delta_r^3/\Delta_w^2 \cup \Delta_w^3 \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \text{ok}}$$

$$\frac{x : T \in \Gamma \qquad T = \mathbf{global}\ T'[] \qquad \Gamma/\Delta_r^1 \vdash e_1 : \text{uint} \qquad \Gamma/\Delta_r^2 \vdash e_2 : T'}{\Gamma/\Delta_r^1 \cup \Delta_r^2/\cdot \vdash a[e_1] := e_2\ \text{ok}}$$

$$\frac{x : T \in \Gamma \qquad T = \mathbf{global}\ T'[] \qquad \Gamma/\Delta_r^1 \vdash e_1 : \mathbf{ID}(a', b', c') \qquad \Gamma/\Delta_r^2 \vdash e_2 : T'}{\Gamma/\Delta_r^1 \cup \Delta_r^2/(a, (a', b', c')) \vdash a[e_1] := e_2\ \text{ok}}$$

$$\frac{\begin{array}{c}\Gamma/\delta_r^1 \vdash e_1 : \text{int} \qquad \Gamma/\delta_r^2 \vdash e_2 : \text{int} \\ \Gamma/\delta_r^3 \vdash e_3 : \text{int} \qquad \Gamma/\Delta_r^0/\Delta_w^0 \vdash S[e_1/x]\ \text{ok} \qquad \Gamma/\Delta_r^1/\Delta_w^1 \vdash S[e_1 + e_3/x]\ \text{ok} \qquad \ldots \qquad \Gamma/\Delta_r^{15}/\Delta_w^{15} \vdash S[e_1 + 15 * e_3/x]\ \text{ok}\end{array}}{\Gamma/\bigcup \Delta_r \cup \bigcup \delta_r/\bigcup \Delta_w \vdash \mathbf{for}\ x\ \mathbf{in}\ [e_1, e_2, e_3]\{S\}\ \text{ok}}$$

The rule for the **for** essentially unrolls the first 16 iterations of the loop. If the loop variable is being used in an array index this rule collects the first 16 values that the index can take. The reason is that the same behavior repeats itself as far as memory coalescing is concerned after the first 16 iterations as the difference in addresses is a multiple of 16 (and there are 16 threads in a half warp)

2.2.3 Statements

$$\frac{\Gamma/\Delta_r^1/\Delta_w^1 \vdash C \text{ ok} \qquad \Gamma/\Delta_r^2/\Delta_w^2 \vdash S \text{ ok}}{\Gamma/\Delta_r^1 \cup \Delta_r^2/\Delta_w^1 \cup \Delta_w^2 \vdash C; S \text{ ok}} \qquad \frac{}{\Gamma/\cdot/\cdot \vdash \langle\rangle \text{ ok}}$$

2.3 Applications

2.3.1 Memory Coalescing

In order to check if an array access is coalesced, we simply substitute 0 through 15 for tid. This tells us the memory locations accessed by a half warp (in fact the first half warp, but the pattern repeats itself for every half warp). Then, the original access was coalesced if the resulting accesses cover an entire memory segment.

2.3.2 Thread/Block Merging

If two neighbouring thread blocks access some memory segments in common, it can be useful to combine the two blocks into one either by merging the two blocks by doubling the number of threads in a block or by merging corresponding threads in neighbouring blocks.

For each array, our system will generate a list of accesses in terms of tid and bid. A second such list is obtained by substituting tid + 1 for tid and bid + 1 for bid (in practice this means converting the access (a, b, c) to $(a, b, c + a + b)$). Then for each list we can determine the memory segments that the accesses lie in (the exact memory locations are not important). If the two lists have some segments in common, then corresponding threads in neighbouring blocks can be merged.

More generally, we can obtain a second list of accesses for each array by substituting bid + 1 for bid. Then, for each list we can determine the *range* of memory segments accessed by that list for all threads in the block. If the ranges for the two lists are overlapping, then this means that neighbouring blocks can be merged.

2.3.3 Kernel Merging

Real GPGPU applications often consist of multiple GPU kernels. These kernels can sometimes access the same array segments. In these cases, merging two kernels into one (by merging corresponding threads) can be beneficial because the shared data can be read from global memory just once by using shared memory as a cache.

Kernel merging has previously been proposed by [4]. However their focus is on reducing power consumption, and not on performance. By identifying shared data segments across kernels, our analysis enables a kernel merging transformation that reduces the total number of global memory accesses.

Consider a pair of kernels which both read from the same array (or arrays) but do not have any data dependencies. Using our method of array access analysis, it is possible to determine when corresponding threads in different kernels access the same memory segments. In this case, a very simple transformation can be used to merge the two kernel into one: simply combine corresponding threads, load the shared segments into shared memory and rewrite global accesses to accesses from the shared memory. This transformation is likely to increase performance since a pair of global memory accesses have now been converted into a single global memory access (and a pair of shared memory accesses).

3 Conclusions and Future Work

We have a devised a type system for a simple GPGPU language that allows us to track array accesses that are linear polynomials of the thread and block indices. We have demonstrated how this can be used to inform the decisions of an optimizing GPGPU compiler. Our present formalism already allows us to analyze some of the GPGPU optimizations found in the literature (for example those in [5]). At the same time, there are many possible directions in which this system could be extended:

- Currently our system deals only with single dimensional arrays and single dimensional thread and block indices. Both these assumptions are quite restrictive in terms of the programs we can analyze. However, we foresee no fundamental difficulty in extending our approach to remove these restrictions.
- Our system allows us to analyze the applicability of one type of kernel merging optimization. However we haven't yet implemented this optimization. Furthermore, we have identified *one* scenario where data sharing information can be used to merge kernels; there may be others. We believe this is a promising avenue for future research.
- Systems such as CUDA-lite [3] rely on programmer supplied annotations in order to analyze program behavior. This approach is complementary to ours. Programmer specified *type* annotations in systems similar to ours may allow more precise and/or more complete analyses.

References

[1] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, 37(3):152–163, 2009.

[2] S.S.B.M.D. Sanjay, J.P.W.D. Gropp, and W.H. Wen-mei. An Adaptive Performance Modeling Tool for GPU Architectures. *Urbana*, 51:61801, 2010.

[3] S.Z. Ueng, M. Lathara, S. Baghsorkhi, and W. Hwu. CUDA-lite: Reducing GPU programming complexity. *Languages and Compilers for Parallel Computing*, pages 1–15, 2008.

[4] G. Wang, Y.S. Lin, and W. Yi. Kernel Fusion: an Effective Method for Better Power Efficiency on Multithreaded GPU. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 344–350. IEEE.

[5] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. *ACM SIGPLAN Notices*, 45(6):86–97, 2010.