Texturizing PPCG : Supporting Texture Memory in a Polyhedral Compiler

*Abhishek Patwardhan
Department of Computer Science and Engineering
IIT Hyderabad, India
Email: cs15mtech11015@iith.ac.in

Ramakrishna Upadrasta
Department of Computer Science and Engineering
IIT Hyderabad, India
Email: ramakrishna@iith.ac.in

***Abstract*—In this paper, we discuss techniques to transform sequential programs to texture/surface memory optimized CUDA programs. We achieve this by using PPCG, an automatic parallelizing compiler based on the Polyhedral model. We implemented a static analysis in PPCG which validates the semantics of the texturized transformed program. Depending on the results of the analysis, our algorithm chooses to use texture and/or surface memory, and alters the Abstract Syntax Tree accordingly. We also modified the code-generation phase of PPCG to take care of various subtleties. We evaluated the texturization algorithm on the PolyBench (4.2.1 beta) benchmark and observed up to 1.6x speedup with a geometric mean of 1.103X. The title and at many places, the paper uses term Texture memory. But, the optimizations are for Texture and Surface memory.**

I. Introduction

The advancements made from the traditional processor architecture to multi-core, many-core machines after the collapse of Moores law are greatly improving the execution times of various applications. Also, special hardware accelerators like GPUs can be used to speedup computations. Due to complexities involved in writing correct and efficient parallel programs on such wide variety of architectures, new compiler technologies are also being proposed which automatically parallelize sequential programs provided as an input to them.

Modern GPU architectures have multiple levels of memory hierarchy. Typically, they also support texture and surface memory, both of which are optimized for array accesses which exhibit spatial locality. A spatial locality is a special type of locality of reference such that, if a program accesses some memory location, then it is very likely that subsequently it would access memory locations that are located in the neighbourhood of the current access location.

With the current widespread use of GPGPU computing, exploiting the usage of these various varieties of memories for applications such as Linear Algebra kernels, Jacobi, Heat-3d equations are also equally important.

Polyhedral model is one of the powerful formalism to represent, analyze and transform programs so as to run efficiently on various modern heterogeneous parallel architectures. PPCG[1] is an open-source automatic parallelizer based on Polyhedral model which transforms input C programs into equivalent CUDA-C/OpenCL programs. PPCG not only extracts and parallelizes the input code but also applies transformations to represent data efficiently onto GPU global memory, shared memory, and registers.

In this paper, we discuss techniques used to augment PPCG which are useful in generating texture memory optimized CUDA code. Remaining part of this paper is organized as follows : First, in section FIXME, we introduce polyhedral model, and then in section FIXME we give brief overview of PPCG. In section FIXME we discuss various aspects of texture and surface memory. In section FIXME we discuss design of texturized PPCG and then in section FIXME we analyze performance and finally in section FIXME we state future work and provide our conclusions.

II. Polyhedral Model

The Polyhedral model focuses on compute-intensive parts of a program and hence targets loop nests. Polyhedral model represents d-deep loop nest appearing in input program as d-dimensional polyhedron in euclidean space. A valid integer point inside a polyhedron represents dynamic instance of loop body. As per model terminology, the compute intensive parts of a program are called Static-Control Part of program (SCoP). Static control parts are essentially statically analyzable because of involvement of affine-expressions for loop bounds,conditionals,array accesses. Once SCoPs are extracted from source program, array data-flow analysis??REF?? is performed. The Analysis detects two iterations of a loop (contained inside a SCoP), both of which access the same array element and at least one iteration writes to that array element.Once dependences are extracted, program transformations must respect them so as to preserve semantics. The scheduler applies various affine transformations onto SCoP which essentially exposes parallelism and improves data locality. Finally, the transformed SCoPs are fed as input to code-generator which replaces old SCoPs with transformed SCoPs. The problem of generating code from Polyhedron wrapped within a SCoP reduces to generating a loop nest which visits all valid integer points within it [2].

III. PPCG

PPCG detects SCoPs from input C programs by using Polyhedral Extraction Tool (PET) [3]. First, PET internally invokes Clang (Open source C frontend) in order to parse input program. PET represents SCoPs in the form of tree referenced hereafter as PET-tree. PPCG performs dependence analysis by using Integer Set Library[4] (ISL) which is the library to represent and manipulate polyhedral integer sets. A dependence analysis implemented in ISL is a classic Feautriers[5] dependence analysis. PPCG implements program transformations by performing equivalent affine transformations over polyhedral integer sets. The scheduling/transformation algorithm implemented inside ISL is the PLuTo[6] algorithm. The vital transformation applied are loop-tiling and loop-permutation both of which help in exposing parallelism and improving data locality. In addition, PPCG applies transformations so as to exploit usage of shared memory, array privatization whenever possible. The transformed polyhedral set is then fed to ISL-code-gen module which generates AST with which CUDA/OpenCL kernel code gets generated. Host-code generator handles data-transfer to/from CPU to GPU, launching of kernel, array allocation/deallocation on global memory.

IV. Texture and Surface Memory

On GPU architectures, Texture and Surface memories are nothing but special type of caches with a sophisticated hardwired address translation mechanism. The central idea behind texture/surface memory is that instead of flattening 2-D or 3-D arrays to have linear access mechanisms, the linearization is done in a way that preserves the original neighborhood of array elements as it would be in 2-D or 3-D space. Doing so enables to cache spatial neighbor elements when a particular array element is accessed.

A. Interpolation mechanism

A natural way to preserve neighborhood of array elements after mapping to linear memory is by approximating space with space-filling curve. Especially Z-curve is widely used for approximation. The specialty of Z-curves lies inside its simplicity of interpolation (which is required in address translation). The interpolation for Z-curve involves application of boolean and shift operations, which can be effectively implemented as a circuit to make it faster.

B. Factors contributing to performance improvement

The two factors contributing to performance improvement by using textures are mentioned below :

- Texture cache is designed in such a way spatial neighbors are located close in linear memory.
- Texture caches are separate from on-chip caches,thereby effectively reducing global memory traffic.

C. Read-Write coherency

An array cannot be written to once it is stored in texture memory. So a major precondition to store an array inside a texture is that it must be read-only. This precondition is too restrictive. However,still there is a work-around for this as mentioned below:

- Create a global memory copy of writable array stored in texture.
- All kernel reads for that array must be from texture.
- All writes to that array must be performed to its global memory copy.

Now this allows a kernel to write to textures safely but with one subtle condition. An updated value must not be read by any of the thread inside kernel.The reason behind is that the updated copy of an array is residing onto global memory and array is read through texture cache. Therefore, it might lead to inconsistent results. This is termed as Read-Write coherency.

D. copy avoidance through cudasurfaces

Beyond R-W coherency, above strategy has another drawback. Consider a scenario, where array is stored in texture memory and kernel may write to it. So a copy must be created on to global memory which corresponds to updated array. Now assume subsequently a new kernel is required to be launched and it needs to read those updated values through texture. Hence, it is required to copy updated array residing into the global memory back to the one stored on texture memory. In worst case, kernel may just update a single array entry but still it is necessary to copy entire array due to lack of programming APIs available in CUDA for accessing cuda-array. It turned out that this in itself was a major performance bottleneck. So NVIDIA GPU architectures with Compute capability $>$ 2.0 supports writing to textures through cudasurfaces. But it should be noted that, Read write coherency is applicable to CUDA-surfaces.

V. Texturizing PPCG

In this section, we discuss implementation aspects of generating texturized CUDA code through PPCG. As a part of design, we exposed following options so as to have flexibility and backward compatibility.

- Texture Pragma: Annotation for candidate arrays to consider to store into texture/surfaces.
- -no-surface memory flag: In case if target GPU have compute capability < 2

Our implementation performs analysis to check for Read-Write coherency. Based on analysis, a decision algorithm chooses to store array into texture or surface memory. The Code-generator takes care of transforming kernel AST such that appropriate texture/surface APIs for CUDA are invoked. We discuss each of these phases in the subsequent subsection.

A. Static analysis

In order to preserve semantics of original program, it is necessary not to store arrays into texture/surface memory for whom updated values are likely to be read again within same kernel. Since, GPU programs are Single Instructions Multiple Data (SIMD), it suffices to ensure that array is not accessed in write followed by Read(W-R) order.So analysis looks for all array access inside a kernel by traversing tree and if it finds W-R order for some array, then analysis invalidates that array. We do static analysis separately for each kernel.The texture cache gets refreshed each time when new kernel is launched, thereby values updated by previous kernel would be read consistently in next kernel launch and hence can be read safely.

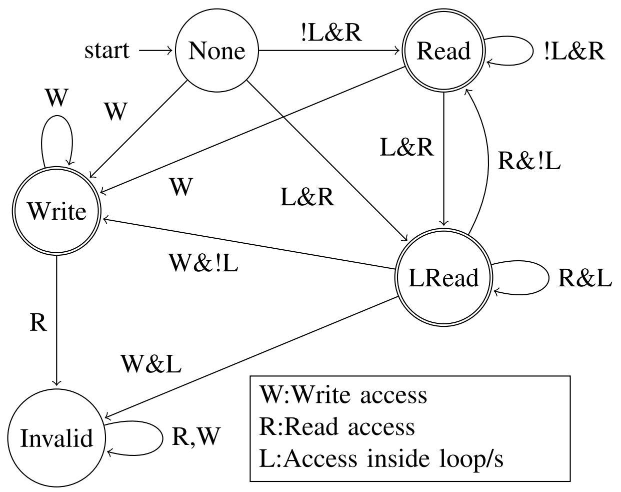

Fig. 1. Automata showing transitions of new_state

A subtle point while traversing tree is that children of a node must be processed in right to left order. The reason in doing so ensures processing of rvalue nodes first and then lvalue in assignment statement.

Another small caveat in static analysis is that if a loop contains Read followed by Write (R-W) order for some array, then it is likely to cause R-W coherency. Because of implicit back-edge for a loop, (R-W) order can effectively result in : R-**W-R**-W-R-W,.... Hence, it is needed to identify R-W order when access is surrounded within a loop nest.

A pseudo-code which performs static analysis is given in Algorithm 1. And corresponding automata which decides how new_state gets computed within Algorithm 1 is shown in figure ??FIXME??.

Algorithm 1 Static analysis

```
Require: PET_tree $\mathcal{T}$, Candidate_arrays $\mathcal{C}$
1: for each Kernel K $\in \mathcal{T}$. kernels do
2:     $\forall$ array A $\in \mathcal{C}$ state(A,K) $\leftarrow$ None
3:     for each Array access a $\in$ K do
4:         if array_name(a) $\notin \mathcal{C}$ continue end if
5:         old_state $\leftarrow$ state(array_name(a),K)
6:         new_state $\leftarrow$ get_new_state(old_state, a.is_write,
           a.is_inside_loop)
7:         state(array_name(a),K) $\leftarrow$ new_state
8:     end for
9: end for
```

B. Decision for using texture or surfaces

For all arrays within each SCoP, a single decision is made about storing them in texture/surface.In this way, texture setup time is avoided during two successive kernel launch belonging to the same SCoP.

Out of all candidate arrays, those end-up being in invalid state arent considered further. If array is not Read-Only within a SCoP then we prefer to store it in surface memory. However, if –no-surface-memory flag is set then we store array into texture, and perform writes onto copy which is residing on global memory.In some cases, PPCG automatically does array linearization so as to achieve coalesced access within a warp. In such cases, we prefer to use linear texture memory over cudaarray bound texture. The decision algorithm is provided in Algorithm 2.

Algorithm 2 Decide texture or surface

```
Require: List of candidate arrays : $\mathcal{C}$
Require: Compute capability $< 2$ : ccpFlag
1: for each Candidate array A $\in \mathcal{C}$ do
2:     $\exists$ kernel k (state(k,A)=invalid)?mark(A,noTex,noSurf)
3:     (A.is_linearized) ? mark(A,Tex,noSurf)
4:     (A.is_Read_only) ? mark(A,Tex,noSurf)
5:     (ccpFlag) ? mark(A,Tex,noSurf) : mark(A,NoTex,Surf)
6: end for
```

C. Code generation

A code-generation phase consists of augmenting PPCG CUDA code-generator so as to take care of following :

- Setup texture,surface references. (Host code-generator)
- Access device arrays through texture/surface references. (kernel code-generator)

1) Host code generation: Host code generation is relatively easy. For each array, based on decision taken by Algorithm 2, we generate a code which

- Declares a texture/surface reference.
- Declares a channel format.
- Declares and allocates cudaArray/Linear texture.
- Copies array from CPU memory to cudaArray.
- Binds cudaArray to texture/surface reference.
- Copies array back from surface memory to CPU memory after all kernels within a SCoP are called.
- Unbinds texture references.

In addition to that, in case if –no-surface-memory flag is provided then after end of kernel launch statement, it is needed to copy updated array values from global memory copy to the cudaarray bound to texture reference. This ensures subsequent kernels read correct values.We generate such copy statements only for the arrays that are likely to be modified by a kernel. This information is already encoded within a data structure which stores static analysis result.

2) Kernel code generation: For kernel code generation AST representing kernel code is traversed and altered based on whether array is in texture or surface and current access is read or write.

If array is residing in texture, then all read accesses for array are replaced with cuda texture read function call. If array is decided to be stored on surface memory then surface Read/Write operation is performed through temporary variable so as to adhere to syntax. An algorithm transforming kernel AST is shown in Algorithm ??FIXME??

Algorithm 3 Transform AST representing kernel code

```
Require: Set of kernel AST's $\mathcal{S}$
 1: for each Kernel tree $\mathcal{T} \in \mathcal{S}$ do
 2:     for each array access A $\in \mathcal{T}$ do
 3:         if array(A).in_texture then
 4:             if A.isRead then
 5:                 transform_to_texture_read_call(A)
 6:             else
 7:                 Do Nothing.
 8:             end if
 9:         else if array(A).in_surface then
10:             if A.isRead then
11:                 tempVar = new_temp()
12:                 gen_stmt(tempVar=gen_surface_read(A.indices))
13:                 transform_access_to_scalar_read(A,tempVar)
14:             else
15:                 tempVar = new_temp()
16:                 gen_stmt(tempVar= get_Rvalue(A))
17:                 generate_surface_write(tempVar,A.indices)
18:             end if
19:         end if
20:     end for
21: end for
```

VI. Performance Evaluation

We evaluated texturized version of PPCG onto PolyBench beta 4.1 which is widely used benchmark in Polyhedral community. For experimentation purpose, we **disabled** usage of shared memory, array privatization and array linearization.

We performed all experiments onto machine with following specification ???FIXME??

We inspected PolyBench kernels for two metrics: correctness and execution time. We evaluated correctness by running each benchmark with small enough data set size and comparing array dumps of texturized version with non-texturized version. For all benchmarks, we found that semantics are preserved, thanks to static analysis.

For recording execution times, all benchmarks were set to their default data set size. Our experimental procedure for determining execution time of each program is given below:

- Run each benchmark five times. Record execution time.
- Eliminate two extremal execution times and verify that deviation for remaining is less than 5%.If not repeat experiment.
- Repeat above steps thrice.
- Take median of these three values.

As a baseline to compare execution times, we recorded execution time of CUDA codes generated by PPCG which doesn't use texture and surface memory.

The graphs comparing execution time for texturized version versus non-texturized version is shown in Figure 2. The corresponding speedup plot is shown in Figure 3

For few benchmarks, we observed that execution times were numerically too small. So it was very likely, that

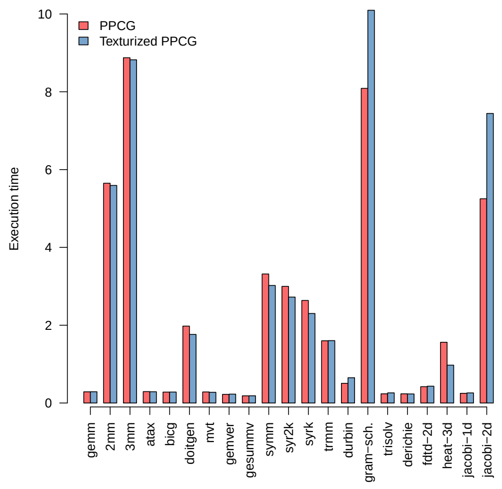

Fig. 2. Program execution time

speedup obtained by using texture/surface memory would get compensated because of extra overhead involved in texture binding/unbinding.Hence, we followed above mentioned experimental procedure and recorded execution times for computation section of code. We discovered that it was worth doing this exercise, as 4 benchmarks were falling into this category. The speedup obtained by CUDA kernels is illustrated graphically in Figure 4.

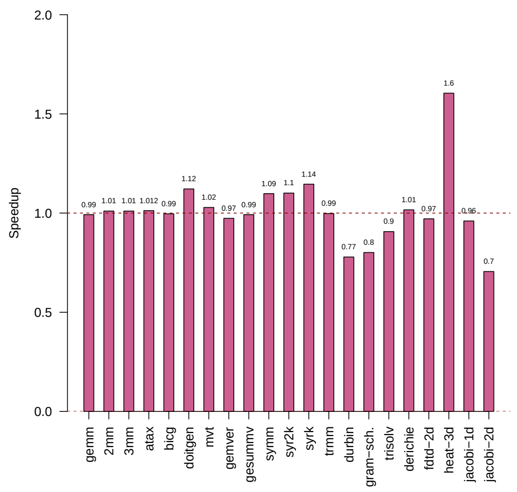

Fig. 3. Speedup

Out of 28 programs, 7 programs were rejected after static analysis and were not texturized. Out of remaining 21 programs, 10 programs showed direct improvement in their execution time. In all 14 programs showed improvement in their kernel execution time. Average speedup observed was 1.114.

Some observations from the evaluations are summarized below :

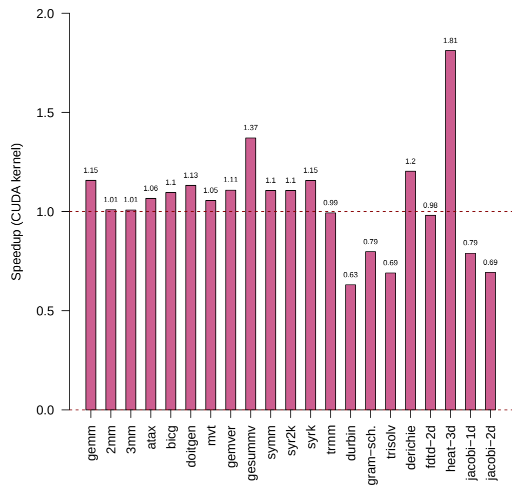

Fig. 4. Speedup for kernels

- Many of linear algebra kernels showed speedup in texturized version.
- 3-D arrays are best candidates to store into texture and surface memory provided access pattern suites.
- Even though jacobi-2d kernel exhibit spatial locality, but only a single array element is reused by two neighboring thread and hence performance was not improved. Similar is the case for jacobi-1d kernel.
- Surface memory must be used sparingly.

VII. Conclusion

We proposed a method to support texture memory and surface memory in the polyhedral compiler PPCG. To the best of our knowledge, this is the first attempt to automatically generate texturized CUDA code, beginning from a C program.

We mainly addressed the problem of generating texturized CUDA code which ensures preservation of the semantics of the input program by performing static analysis. We evaluated the generated code against the non-texturized CUDA code for various applications and have shown small, but significant speedup for programs belonging to domains other than image processing. Thus, we believe, that our contribution is towards increasing the scope of texture and surface memories and to the bigger goal of effective GPGPU computing.

VIII. Future work

We believe that much of potential of texture and surface memories is yet to be explored. In future we would like to come up with a cost model which guides usage of the texture and surface memories based on the dependence analysis information provided by polyhedral compilers like PPCG. Also, our current work focuses on CUDA code generation, but, we would like to exploit architectures of other GPU vendors like AMD, and support them through OpenCL. We also believe that more sophisticated static analyses could be performed which exposes cases where R-W coherency does not hold. We believe that the basic infrastructure that we have developed provides an ideal platform for exploring these important aspects.

Acknowledgment

The authors would like to thank Dr. Sven Verdoolaege, Dr. Michael Kruse, Prof (Dr). Albert Cohen for their motivation, feedback, and guidance.

References

[1] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013.

[2] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 7–16. [Online]. Available: http://dx.doi.org/10.1109/PACT.2004.11

[3] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, Jan. 2012.

[4] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Mathematical Software - ICMS 2010*, ser. Lecture Notes in Computer Science, K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, Eds. Springer, 2010, vol. 6327, pp. 299–302.

[5] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991. [Online]. Available: http://dx.doi.org/10.1007/BF01407931

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595