Automatic Communication Optimizations Through Memory Reuse Strategies

Muthu Baskaran Nicolas Vasilache Benoit Meister Richard Lethin

Reservoir Labs Inc.
632 Broadway, New York, NY, USA
{baskaran,vasilache,meister,lethin}@reservoir.com

Abstract

Modern parallel architectures are emerging with sophisticated hardware consisting of hierarchically placed parallel processors and memories. The properties of memories in a system vary wildly, not only quantitatively (size, latency, bandwidth, number of banks) but also qualitatively (scratchpad, cache). Along with the emergence of such architectures comes the need for effectively utilizing the parallel processors and properly managing data movement across memories to improve memory bandwidth and hide data transfer latency. In this paper, we describe some of the high-level optimizations that are targeted at the improvement of memory performance in the R-Stream compiler, a high-level source-to-source automatic parallelizing compiler. We direct our focus in this paper on optimizing communications (data transfers) by improving memory reuse at various levels of an explicit memory hierarchy. This general concept is well-suited to the hardware properties of GPGPUs, which is the architecture that we concentrate on for this paper. We apply our techniques and obtain performance improvement on various stencil kernels including an important iterative stencil kernel in seismic processing applications where the performance is comparable to that of the state-of-the-art implementation of the kernel by a CUDA expert.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization

General Terms Algorithms, Performance

Keywords Data reuse, Memory reuse, Data transfer optimization

1. Introduction

The tension between parallelism and locality of memory references is an important topic for high-performance application optimization. More parallelism allows more concurrent execution of the parallel portions of a program. Increasing locality directly translates into communication reduction between memories and processing elements. Increasing parallelism may decrease locality and vice-versa. This presents a huge challenge to achieve good performance in modern multi-node and multi-socket architectures. One such example is the complexity in achieving good performance on GPGPUs. In a first approximation, simple translation from C to CUDA yields functional correctness. However performance is likely to be low. This is because only accesses to contiguous elements in memory result in efficient memory transactions. As soon as memory accesses are not contiguous, it is recommended to use explicit memory transfers into shared memory. Additionally, if temporal reuse can be exhibited, it is often recommended to use explicit communications into shared memory. On top of this, private memory exhibits the lowest access latencies but is not accessible by neighboring threads. The goal of the R-Stream compiler is to help solve all these complex implementation trade-offs in a generalized framework. R-Stream provides an automatic source-to-source mapping pathway from a high-level textbook-style code expressed in ANSI C to a target-specific source.

In this paper, we focus our attention on a specific capability of R-Stream , namely, optimizing communications by improving memory reuse at various levels of an explicit memory hierarchy. We pick GPGPUs as our target architecture, as these memory performance improvement techniques can be well-illustrated with the hardware properties of GPGPUs. We apply our techniques and show performance improvement on various stencil kernels and an important iterative stencil kernel in seismic processing applications, namely, the Discretized Wave Equation (DWE) kernel used in Reverse Time Migration algorithm of seismic computing. The performance of our optimized DWE kernel is within 23% of previously published hand-tuned results by an expert programmer [1].

2. Details of Communication Optimizations

Our emphasis in this work is on the memory reuse optimizations in GPUs that particularly focus on managing on-chip memories such as the shared memory and registers. For clearer explanation of the concepts behind the optimizations, we run through the optimizations over the DWE kernel. The DWE is a 3D 8th order stencil (eight input elements in each dimension, not counting the center stencil point). We start our discussion from the following CUDA code excerpt that is automatically generated by R-Stream after applying a 2D tiling (with tile sizes of 8×32) and a 2D thread and thread block distribution on the DWE kernel.

```
float __shared__ U2_1[16][40];
for (i = 0; i <= 255; i++) {
      int ix = 32 * blockIdx.x + threadIdx.x;
      int iy = 8 * blockIdx.y + threadIdx.y;
      /* No register reuse */
      float U2_l_5 = U2_3[i][4 + iy][4 + ix];
      float U2_l_4 = U2_3[1+i][4 + iy][4 + ix];
      float U2_l_1 = U2_3[2+i][4 + iy][4 + ix];
```


PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

```
      float U2_1_2 = U2_3[3+i][4 + iy][4 + ix];
      float U2_1_9 = U2_3[4+i][4 + iy][4 + ix];
      float U2_1_3 = U2_3[5+i][4 + iy][4 + ix];
      float U2_1_6 = U2_3[6+i][4 + iy][4 + ix];
      float U2_1_8 = U2_3[7+i][4 + iy][4 + ix];
      float U2_1_7 = U2_3[8+i][4 + iy][4 + ix];

      /* No shared memory optimization */
      #pragma unroll
      for (k = 0; k <= 1; k++){
       #pragma unroll
       for (j = 0; j <=(threadIdx.x+39 >>5); j++){
         U2_1[8 * k + threadIdx.y]
                  [32 * j + threadIdx.x] =
         U2_3[4 + i][8 * k + iy][32 * j + ix];
       }
      }
       ...
}
```

2.1 Register Reuse and Rotation

We define "register reuse/rotation" as the ability to move data into registers and reuse it and also to rotate data from one register to another for reusing it across various loop iterations. While there are various works in literature that target latency hiding in GPUs by moving data into registers and reusing the data, we are not aware of any prior work that automatically achieves the capability to rotate data from one register to another across loop iterations.

In the above code excerpt, data is moved into registers for each iteration of the i loop. However there is a clear reuse of data between different registers from one iteration of the loop to the next iteration. This is automatically exploited by R-Stream.

2.2 Shared Memory Optimization

Usual GPU optimizations involve moving data that is shared across various threads in a thread block into shared memory and reusing the data. Our "shared memory optimization" is not limited to storing and reusing shared data, but also, moving and reusing data from registers to shared memory and vice versa to reduce the number of global memory accesses and reduce communication latency and improve memory bandwidth.

The optimized code resulting from these optimizations from R-Stream looks like:

```
float __shared__ U2_1[16][40];
for(i = 0; i <= 255; i++) {
...
      int ix = 32 * blockIdx.x + threadIdx.x;
      int iy = 8 * blockIdx.y + threadIdx.y;
      /* Register reuse optimization */
      if (i == 0) {
         U2_1_5 = U2_3[0][4 + iy][4 + ix];
         U2_1_4 = U2_3[1][4 + iy][4 + ix];
         U2_1_1 = U2_3[2][4 + iy][4 + ix];
         U2_1_2 = U2_3[3][4 + iy][4 + ix];
         U2_1_9 = U2_3[4][4 + iy][4 + ix];
         U2_1_3 = U2_3[5][4 + iy][4 + ix];
         U2_1_6 = U2_3[6][4 + iy][4 + ix];
         U2_1_8 = U2_3[7][4 + iy][4 + ix];
         U2_1_7 = U2_3[8][4 + iy][4 + ix];
      }
      if (i >= 1) {
         U2_1_5 = U2_1_4;
         U2_1_4 = U2_1_1;
         U2_1_1 = U2_1_2;
         U2_1_2 = U2_1_9;
         U2_1_9 = U2_1_3;
         U2_1_3 = U2_1_6;
         U2_1_6 = U2_1_8;
         U2_1_8 = U2_1_7;
         U2_1_7 = U2_3[8 + i][4 + iy][4 + ix];
      }
       ...
      /* Shared memory opt - Filling from reg. */
      U2_1[threadIdx.y + 4]
           [threadIdx.x + 4] = U2_1_9;
       ...
}
```

2.3 Brief Discussion on the Method

In general, R-Stream compiler tries to move data to faster memories as much as possible and reuse data from faster memories to perform computation. The basic idea behind the determination of the feasibility of these optimizations is described below.

The first step is to find pairs of data transfers such that (1) they are executed within the same processing element (at the memory accessible level), (2) both involve reads from the same global memory location, and (3) both involve writes to a faster on-chip memory. The next step is to check if the value read from the first global memory read is "live" in a faster memory (and is not possibly flushed/overwritten) till the point of execution of the second global memory read. If it is the case, then replace the second global memory read with a read from the faster memory.

3. Performance Evaluation

We present our experimental results on two different GPU chips, namely, (1) GTX 285 (Tesla GPU) and (2) GTX 480 (Fermi GPU). We evaluate our techniques on three different types of stencils - (a) DWE kernel, (b) out-of-place Jacobi stencil (a 3-D 7 points stencil), and (c) finite-difference (FD) stencil used in acoustic modeling. The volume of the stencils evaluated is $256 \times 256 \times 256$. Table 1 presents the results of the stencil kernels. There are two baselines for comparison: (1) a "base" mapping, which uses good tile sizes and good thread and block distribution but none of the optimizations discussed in the paper, and (2) a hand-optimized "expert" version (only for DWE kernel) following the guidelines of [1]. It can be observed that the remaining speedup between the best DWE code generated automatically by our compiler and the hand-tuned DWE code is of 23% on Tesla and 10.2% on Fermi.

Version	DWE		Jacobi		FD	
	Tesla	Fermi	Tesla	Fermi	Tesla	Fermi
Base	12.91	7.79	6.6	2.5	6.04	4.7
R-Stream	5.99	4.68	4.10	2.21	4.74	3.50
Expert	4.58	4.20	-	-	-	-

Table 1. Execution time of 3D stencil kernels (in ms) on Tesla and Fermi GPUs

4. Conclusion

In this paper, we emphasized the importance of developing techniques to improve memory performance and developed communication or data transfer optimizations through various memory reuse strategies. We applied our techniques and showed performance improvement on various stencil kernels.

References

[1] Paulius Micikevicius. 3D Finite Difference Computation on GPUs using CUDA. In *Second Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU-2*, March 2009.