GpuC: Data parallel language extension to CUDA

Zeki Bozkus

Department of Computer Engineering
Kadir Has University, Istanbul, 34083 Turkey

Abstract

In recent years, Graphics Processing Units (GPUs) have emerged as a powerful accelerator for general-purpose computations. Current approaches to program GPUs are still relatively low-level programming models such as Compute Unified Device Architecture (CUDA), a programming model from NVIDIA, and Open Compute Language (OpenCL), created by Apple in cooperation with others. These two programming models have all the complexity of parallel programming such as breaking up the task into smaller tasks, assigning the smaller tasks to multiple CPUs to work on simultaneously, and coordinating the CPUs. There is a growing need to lower the complexity of programming these devices. In this paper, we propose a data-parallel loop (forall) extension to the CUDA programming model. We describe our prototype compiler named GpuC.

Keywords: GPGPU, Data Parallel, CUDA, Compiler Optimizations

1. Introduction

We used GCC, which is an open-source compiler that is used by many researchers as the platform for implementing application-specific compilers. We added an extension to GCC to accept forall loops and the other CUDA constructs. Forall loop computations are more suitable for the Single Instruction Multiple Data (SIMD) model of the GPU architecture.

The compiler performs source-to-source translation from CUDA with data-parallel loops (forall) to CUDA C language for NVIDIA GPU. This programming model will make GPU programming accessible and allow many real-world applications that are easily implemented on GPUs to run significantly faster than on the regular multi-core systems without GPU. The results will help many other interdisciplinary applications such as molecular simulations, computational chemistry, and medical imaging.

2. Forall statement

High Performance Fortran([9], [10]) introduced a statement forall as an alternative to the DO-loop. The intent of this statement was that its contents can be executed in any order, independent of the index. It therefore gives the possibility of a parallel implementation. GpuC has adapted a similar forall statement with C style syntax as shown in Figure 1.(a) However, some rules are introduced in order to avoid side effects, such as function calls inside forall cannot be recursive and cannot have side effects on global data or its arguments. Figure 1.(b) shows a simple example of forall which inverts the elements of a matrix, avoiding division with zero.

![Figure 1: Data parallel forall statements](image)

GpuC tries to execute forall statements on GPU device by efficient parallel implementation. If, for some reason, it can not execute on GPU, the compiler informs the user with a warning attached with a reason at compile time. In this sequential execution case, the compiler replaces the forall with a series of regular for statements. The forall can have a scalar variables at the left hand side assignment. GpuC can analyze these cases and recognize the reduction pattern which will be explained more detail on Section 3.

3. Reduction recognition at forall

An important aspect of data parallelism is the reduction operation where a reduction function computes a scalar value from an array. The reduction operation is a simple and powerful parallel primitive with a broad range of application. Table 1 shows a number of reduction functions and explains their mathematical definition. Many parallel programming paradigms feature a set of reduction operations, among them are HPF [10], OpenMP [2], and Accelerator [9]. These parallel languages provide explicit features to express reduction operation such as function calls or compiler directive with a reduction clause as in the case of OpenMP. However, GpuC will try to detect reduction pattern in forall.
loops by recognizing certain patterns of assignments from arrays to a scalar variables.

<table>
<thead>
<tr>
<th>Reduction function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM(A)</td>
<td>$\sum_{i=0}^{N} A(i)$</td>
</tr>
<tr>
<td>PRODUCT(A)</td>
<td>$\prod_{i=0}^{N} A(i)$</td>
</tr>
<tr>
<td>DOTPRODUCT(A,B)</td>
<td>$\sum_{i=0}^{N} (A(i)xB(i))$</td>
</tr>
<tr>
<td>MAXVAL(A)</td>
<td>$\max_{i=0}^{N} A(i)$</td>
</tr>
<tr>
<td>MINVAL(A)</td>
<td>$\min_{i=0}^{N} A(i)$</td>
</tr>
</tbody>
</table>

**Table 1:** show reduction functions and definitions where array $A[N]$, $B[N]$.

Table 2 specifies a mapping used by GpuC between forall code patterns and the reductions they represent. GpuC detects the patterns and calls the corresponding predefined functions. These functions can only be applied to a few datatypes such as int, and float. When an array is passed as an argument to some of reduction primitives, it is also necessary to provide information such as its shape, size, dimension and type etc. All this information is stored into structure which is called array descriptor type (ADT).

To detect the reductions patterns of Table 2, our compiler performs a data flow analysis (DFA). The compiler finds a set of scalar and array variables read in the forall statement (USES) and a set of scalar variable written in the forall (DEFS). It then uses this DFA information to find matching patterns of Table 2.

<table>
<thead>
<tr>
<th>Reduction Pattern</th>
<th>Reduction run-time library calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R = 0.0$; forall(int i = 0; i &lt; N; i++) $R = R + A[i]$;</td>
<td>Reduce_sum_float(float *A, ADT A_shape, float R);</td>
</tr>
<tr>
<td>$R = 1.0$; forall(int i = 0; i &lt; N; i++) $R = R * A[i]$;</td>
<td>Reduce_product_float(float *A, ADT A_shape, float R);</td>
</tr>
<tr>
<td>$R = 0.0$; forall(int i = 0; i &lt; N; i++) $R = R + A[i] * B[i]$;</td>
<td>Reduce_dotproduct_float(float *A, ADT A_shape, float *B, ADT B_shape, float R);</td>
</tr>
<tr>
<td>$R = \text{MIN FLOAT}$, forall(int i = 0; i &lt; N; i++) if($A[i] &gt; R$) $R = A[i]$;</td>
<td>Reduce_maxvalue_float(float *A, ADT A_shape, float R);</td>
</tr>
<tr>
<td>$R = \text{MIN FLOAT}$, forall(int i = 0; i &lt; N; i++) if($A[i] &lt; R$) $R = A[i]$;</td>
<td>Reduce_minvalue_float(float *A, ADT A_shape, float R);</td>
</tr>
</tbody>
</table>

**Table 2:** mapping forall patterns to corresponding reduction run-time library routines.

Our reduction functions were implemented using the algorithm presented by Haris [14]. He presented an algorithm to calculate parallel scan operations by using balanced trees on GPU. The presented scan algorithm consists of two phases; the reduce phase and the down sweep phase. The first phase is known as a parallel reduction. The phase is enough to calculate the reduction function. The CUDA code is available online [4]. The code is implemented on shared memory of GPU.

### 4. Forall kernel definition at the device

The kernel function should handle the distribution of the computations across thread blocks and across threads within a thread block. The aim of the generated code is to optimize memory accesses whether data is in the global memory or it is in the shared memory of a thread block. Figure 7 gives the generated code of the forall at Figure 4.a.

Each thread block in the grid must have the same number of threads. Depending on the array sizes, this may result into excess threads that do not have data to operate on.

```c
#define tile_x 32
#define tile_y 32
__global__ void forall_loop1_lhs_dim1_kernel( float A_device,
                                          float B_device,
                                          float C_device)
{  
  // STEP 1: shared memory declaration with tile
  __shared__ float a_shared[tile_x][tile_y];
  __shared__ float b_shared[tile_x][tile_y];
  __shared__ float c_shared[tile_x][tile_y];
  // STEP 2: calculate the number of tile iteration
  int ntile_x = PROBLEM_SIZE_X / tile_x + 1;
  // STEP 3: Start tiling
  for(int i = 0; i < ntile_x; i++) {
    // STEP 4: load inputs to the shared memory tiles by coalescing
    coalesced_copy(b_device, b_shared,...);
    coalesced_copy(c_device, c_shared,...);
    // STEP 5: perform the data parallel kernel computation
    for(int i=0; i < tile_y; i++ ) {
      int glb_idx = local_to_global(i,tile_y,
                                  NBLOCK1_X,NTHREAD1_X,tid,bid);
      a_shared[thid][i]=
       b_shared[thid][i]+c_shared[thid][i];
    }
    // STEP 6: Copy results back to the global dim
    coalesced_copy(a_shared,a_device,...);
  }
}
```

**Figure 7:** Generated code of kernel definition at the device for forall at Figure 4.a.

### 5. Related work

Accelerator project [9] has a similar goal as our compiler. Instead of parallel loop, Accelerator is an
array-based language based on C# and all computation is done via operations on arrays. It provides a higher-level programming model. Programmers do not have to divide computations into kernels and no aspect of GPUs is exposed to programmer. The Accelerator compiles the data-parallel operations on the optimized GPU pixel shader code.

OpenMP-to-GPGPU [8] compiler translates OpenMp shared-memory programs into CUDA-based GPGPU programs. This work identifies several key transformation techniques to enable efficient GPU global memory access such as loop-swap and matrix transpose techniques for regular applications and loop collapsing for irregular ones. Our approach similar to OpenMP-to-GPGPU compiler that we also support a familiar loop level parallelism. By contrast, our compiler uses tiling technique to explore the efficiency of the shared memory on GPU. We have different approaches to map the computation on the GPU. In addition, our compiler automatically recognize the reduction on the forall statement.

Compile-time transformation techniques [12] have been proposed to address critical performance influencing factors on GPUs. The proposed work optimizes affine loop nests using a polyhedral compiler model. This mathematical compiler model of data dependence is used for efficient data access from GPU global memory and parallelizes the nested loops. Our compiler does not perform data dependence analyzes. Parallelism is explicit with the forall construct. This work also copies the portion of the data to a tile at the shared memory to benefit for the data reuse. However, we copy data from global memory to shared memory even if there is no data reuse to get the performance benefit of coalescing global memory accesses on the GPU architecture. Moreover, we have developed gather and scatter run-time routines for irregular problems since the array subscripts are only known at run-time.

CUDA-lite [13] provides special annotations not a compiler to help programmers to optimize memory performance under CUDA programming environment for GPU. CUDA-lite generates codes for optimal tiling of global memory data. This is an important task due to the large effect memory performance has on overall performance. Our compiler uses similar memory optimizations as CUDA-lite. However, our compiler automatically extracts the memory optimizations as well as generates code to parallelize.

6. Conclusion

In this paper, we proposed a data parallel forall loop as an extension of CUDA programming languages. We showed the compilation steps. The result is a more high-level programming model that allows its users to take full advantage of the GPU’s powerful hardware. We are working to present experimental results from four NAS benchmarks to show performance gains at our proposed GpuC data parallel language.

12. References