CUDA-lite: Reducing GPU Programming Complexity

Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{ueng, mlathara, bsadeghi, hwu}@crhc.uiuc.edu

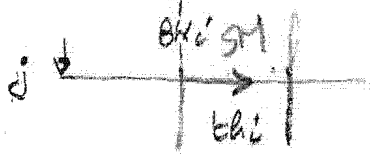
Abstract. The computer industry has transitioned into multi-core and many-core parallel systems. The CUDA programming environment from NVIDIA is an attempt to make programming many-core GPUs more accessible to programmers. However, there are still many burdens placed upon the programmer to maximize performance when using CUDA. One such burden is dealing with the complex memory hierarchy. Efficient and correct usage of the various memories is essential, making a difference of 2-17x in performance. Currently, the task of determining the appropriate memory to use and the coding of data transfer between memories is still left to the programmer. We believe that this task can be better performed by automated tools. We present CUDA-lite, an enhancement to CUDA, as one such tool. We leverage programmer knowledge via annotations to perform transformations and show preliminary results that indicate auto-generated code can have performance comparable to hand coding.

1 Introduction

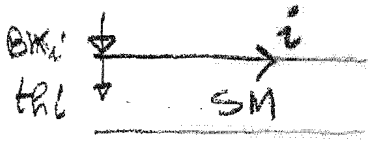
In 2007, NVIDIA introduced the Compute Unified Device Architecture (CUDA) [9], an extended ANSI C programming model. Under CUDA, Graphics Processing Units (GPUs) consist of many processor cores, each of which can directly address into a global memory. This allows for a much more flexible programming model than previous GPGPU programming models [11], and allows developers to implement a wider variety of data-parallel kernels. As a result, CUDA has rapidly gained acceptance in application domains where GPUs are used to execute compute intensive, data-parallel application kernels.

While GPUs have been designed with higher memory bandwidth than CPUs, the even higher compute throughput of GPUs can easily saturate their available memory bandwidth. For example, the NVIDIA GeForce 8800 GTX comes with 86.4 GB/s memory bandwidth, approximately ten times that of Intel CPUs on a Front Side Bus. However, since the GeForce 8800 has a peak performance of 384 GFLOPS and each floating point operation operates on up to 12 bytes of source data, the available memory bandwidth cannot sustain even a small fraction of the peak performance if all of the source data are accessed from global memory.

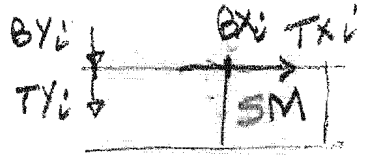
Mapping possibilities: Row-Major EA = $j \times Asize + i$



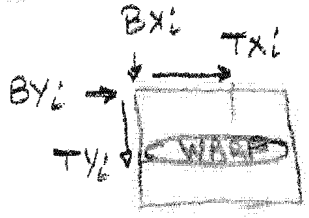
1- $j \cdot Asize + BK_i \cdot TPB + th_i$ (1D-Block, i)
⇒ Coalesced, $\alpha_{i\,(Asize)}$

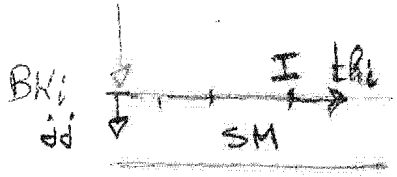


2- $(BK_i * TPB + th_i) Asize + i$ (1D-Block, j)
⇒ Non-Coalesced, $\alpha_{i\,(Asize)}$

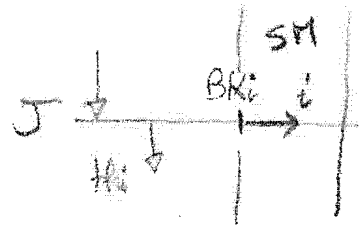


3- $(BY_i \cdot TPB + TY_i) Asize + BX_i \cdot TPB + TX_i$ (2D-Block, i, j)
⇒ No loop, Coalesced, if row-major warp





1'- $(BK_i \times TPB + dd) Asize + I \times TPB + th_i$ (1D-Block, dd)
⇒ Coalesced, $\alpha_{dd,\,I\,(TPB \times TPB)}$



1''- $(J \times TPB + th_i) Asize + BK_i \cdot TPB + i$ (1D-Block, dd)
⇒ Non Coalesced, $\alpha_{J,i\,(TPB \times TPB)}$

Notes:
(1) All 1D-Block lead to a thread having to visit Asize or TPB×TPB. Th will visit all the remaining dimension (th granule size)

```
1   #define ASIZE 3000
    #define TPB 256

    __global__ void
5   kernel (float *a, float *b)
    {
      int thi = threadIdx.x;
      int bki = blockIdx.x;
      float t = (float) thi + bki;
10    int i;

      if (bki * TPB + thi >= ASIZE)
        return;

15    for (i = 0; i < ASIZE; i++)
        {
          b[(bki*TPB+thi)*ASIZE + i] =
            a[(bki*TPB+thi)*ASIZE + i] * t;
        }
20  }
```

```
int main ()
{
  int num_blocks;
  int size = sizeof (float) * ASIZE * ASIZE;

  /* Allocate a_host and b_host,
   * and initialize a_host with values */

  /* Allocate a_device and b_device */
  cudaMalloc ((void **) &a_device, size);
  cudaMalloc ((void **) &b_device, size);

  /* Copy values from host to device */
  cudaMemcpy (a_device, a_host, size,
              cudaMemcpyHostToDevice);

  num_blocks = ASIZE % TPB == 0 ?
    ASIZE / TPB : (ASIZE / TPB) + 1;

  /* Number of thread blocks in the grid */
  dim3 gridDim (num_blocks);
  /* Number of threads per thread block */
  dim3 blockDim (TPB);

  /* Start executing on the GPU */
  kernel <<<gridDim, blockDim>>>
    (a_device, b_device);

  /* Copy values from device back to host */
  cudaMemcpy (b_host, b_device, size,
              cudaMemcpyDeviceToHost);
}
```

Fig. 2. Example Code: Base Case

general-purpose applications to be ported easily onto the GPU. A straightforward implementation of an application would be to utilize only global memory as a proof of concept for parallelizing the algorithm on CUDA.

Figure 2 shows an example CUDA code. The function `main` sets up the data for computation on the CPU while the function `kernel` contains the code that is actually executed on the GPU. Notice that variables that reside in the global memory of the GPU, like `a_device`, are allocated in `main` and data movement is also performed there via API calls to `cudaMemcpy`.

In the `kernel` function, each thread on the GPU traverses a different row of the 2-D array `a`, scaling each element by a thread specific value before storing into the corresponding location in array `b`. Since each TB must have the same number of threads, depending on the data size and program parallelization there may be excess threads that do not have data to operate on. The conditional check on line 12 that exits the kernel function before the loop handles these cases. This check becomes important as we attempt to utilize memory coalescing (Section 2.3).

2.2 Shared Memory

Shared memory is a small (16KB per SM for the GeForce 8800) readable and writable on-chip memory and as fast as register access. Shared memory is uninitialized at the beginning of execution, and resident data is private to each TB

3(c). The tile is first traversed along the column and data is coalesced loaded into a buffer in shared memory, indicated by the grayed arrows. The algorithm then operates on the data along the row from shared memory before moving to the next tile. The performance improvement from doing coalesced loads and using shared memory makes this worthwhile despite the instruction overhead.

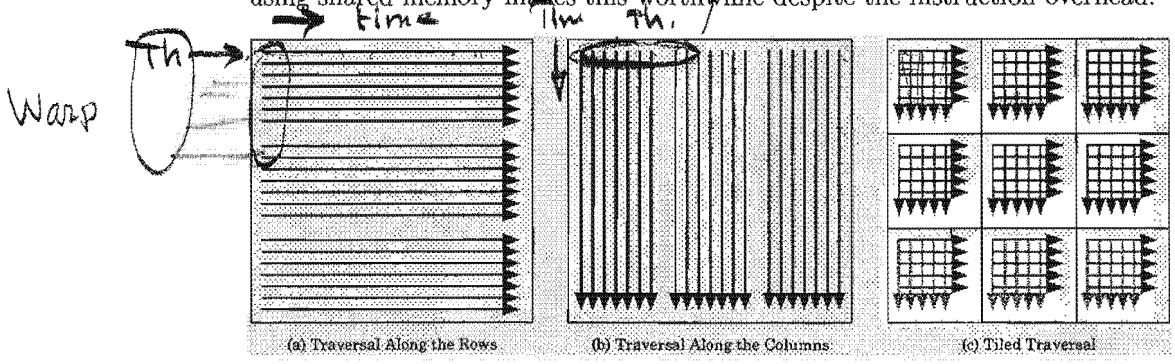


Fig. 3. Graphical View of Data Traversal: (a) Row (b) Column (c) Tiled

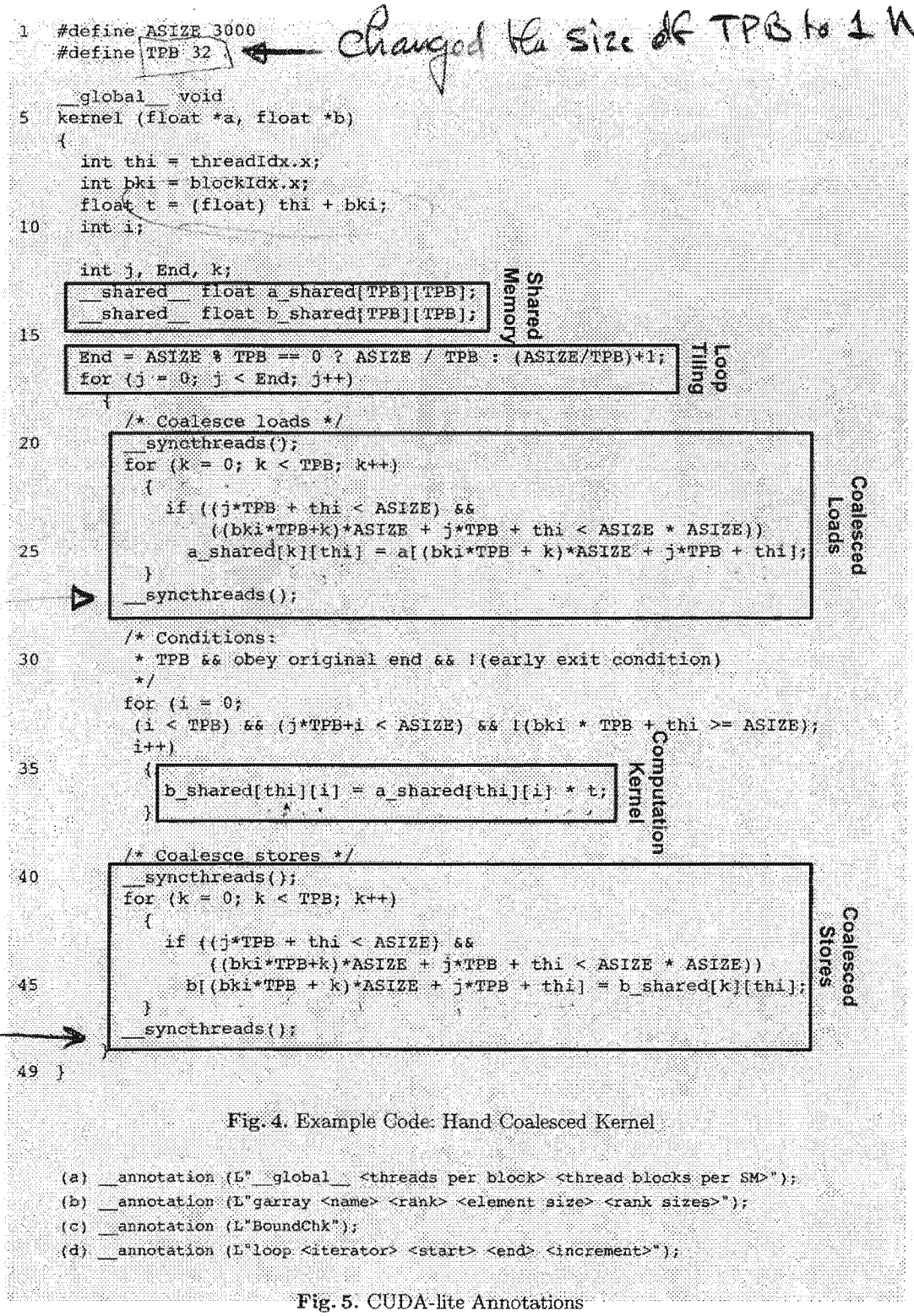
For example, the memory access to array a on line 18 of Figure 2 does not coalesce because it violates rule number 2. For each iteration of the loop, thread `N` accesses `a[N*ASIZE + i]`; `bki` does not matter since the threads are in the same thread block. This means that each thread is accessing data vertically adjacent to each other, as in Figure 3(a), which does not trigger coalescing.

Figure 4 shows the kernel code from Figure 2 rewritten by hand so the algorithm is tiled and the memory accesses coalesced. The amount of code is roughly doubled. The original loop has been tiled and additional code is inserted to load/store data between global and shared memory. The load from array a on line 25 is coalesced since thread `N` accesses `a[k*ASIZE + N]` on each iteration. The computation kernel now operates on the data in shared memory, and the loop around it has included the check on line 12 of the original code as an additional condition. In other words, the excess threads we mentioned back in Section 2.1 may be used to perform memory coalescing accesses, but must not be allowed to perform actual computation.

This rewriting is a large additional burden on the programmer. Not only must the programmer fulfill the memory coalescing requirements, the programmer also has to maintain correctness. The performance improvement this optimization provides will be the ideal, or oracle, case for CUDA-lite.

3 CUDA-lite

Since the behavior of memory coalescing is complex yet understood, we believe that such transformations are best undertaken by an automated tool. This would reduce the potential for errors in writing memory coalescing code, and reduce the burden upon programmers. In our vision, programmers would provide a straight-




```
1   #define ASIZE 3000
    #define TPB 32

    __global__ void
5   kernel (float *a, float *b)
    {
      int thi = threadIdx.x;
      int bki = blockIdx.x;
      float t = (float) thi + bki;
10    int i;

      int j, End, k;
      __shared__ float a_shared[TPB][TPB];
      __shared__ float b_shared[TPB][TPB];
15
      End = ASIZE % TPB == 0 ? ASIZE / TPB : (ASIZE/TPB)+1;
      for (j = 0; j < End; j++)
        {
        /* Coalesce loads */
20      __syncthreads();
        for (k = 0; k < TPB; k++)
          {
            if ((j*TPB + thi < ASIZE) &&
                ((bki*TPB+k)*ASIZE + j*TPB + thi < ASIZE * ASIZE))
25            a_shared[k][thi] = a[(bki*TPB + k)*ASIZE + j*TPB + thi];
          }
        __syncthreads();

        /* Conditions:
30       * TPB && obey original end && !(early exit condition)
         */
        for (i = 0;
         (i < TPB) && (j*TPB+i < ASIZE) && !(bki * TPB + thi >= ASIZE);
         i++)
35       {
           b_shared[thi][i] = a_shared[thi][i] * t;
         }

        /* Coalesce stores */
40      __syncthreads();
        for (k = 0; k < TPB; k++)
          {
            if ((j*TPB + thi < ASIZE) &&
                ((bki*TPB+k)*ASIZE + j*TPB + thi < ASIZE * ASIZE))
45            b[(bki*TPB + k)*ASIZE + j*TPB + thi] = b_shared[k][thi];
          }
        __syncthreads();
      }
49  }
```

Fig. 4. Example Code: Hand Coalesced Kernel

```
(a) __annotation (L"__global__ <threads per block> <thread blocks per SM>");
(b) __annotation (L"garray <name> <rank> <element size> <rank sizes>");
(c) __annotation (L"BoundChk");
(d) __annotation (L"loop <iterator> <start> <end> <increment>");
```

Fig. 5. CUDA-lite Annotations