LSU EE 7722 Homework 4 Due: 19 April 2017

Problem 0: Read Kim et al, "Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures," International Symposium on Computer Architecture, 2013. This paper describes a technique to reduce the wasteful duplication of storage and computation in NVIDIA-style GPUs when each thread in a warp performs the exact same computation or a computation that's a linear function of the thread index. For example,

```
  int num_threads = blockDim.x * gridDim.x;   // Same for all threads.
  int inc = num_threads / N;                        // Same for all threads.
  int tid = threadIdx.x + blockIdx.x * blockDim.x;  // Function of threadIdx
  float eltval = data_in[tid];
```

The idea is to use a special kind of register file, the *ASRF*, in which there is one register per warp rather than one register per thread. Notice that in the example above all threads in a block will have the same value for `num_threads` and `inc`, so these can use the ASRF. The value of `tid` can also be put in the ASRF because its value is a constant plus the thread index. The ASRF cannot be used for `eltval` because the value will be different for each thread.

Problem 1: Appearing below are lines from routine `mxm_volk` in Homework 3. (The entire routine appears at the end of this assignment.) For each one indicate whether it can use the ASRF. If it cannot use the ASRF indicate why not.

(*a*) Comment on whether the ASRF can be used for the values assigned to the three variables below. Assume that `N` is a compile-time constant. *Hint: These three are easy.*

```
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int start = tid / N;
  const int stop = d_app.n_mats;
```

(*b*) Comment on whether the ASRF as described in Kim 13 can be used for the `c0` in the code below for different cases of compile-time constant `CS`. The two easy cases are `CS=1` and `CS` is greater than or equal to the maximum block size. Explain why the ASRF can't be used for `c0` when the value of `CS` is between the two easy cases.

```
  // Column offset to load when populating shared memory.
  //
  const int c0 = threadIdx.x % CS;
```

Problem 2: Consider the lines below from `mxm_volk`:

```
  const int cb = threadIdx.x % N;
  const int c0 = threadIdx.x % C;
  const int h0 = threadIdx.x / N;
  const int b0 = threadIdx.x / C;
```

The moduli and division in the code above are valuable because they divide threads into groups such that threads within a group operate on consecutive items while there may be a gap between items operated on by different groups.

Alas, these are not eligible for the ASRF (except for special values of `N` and `C`) as described by Kim. But clearly they result in exploitable value structure. In this problem we will consider modifications to the ASRF and other hardware operating on affine values so that the value structure in the code above can be exploited.

We'll start with the name, since the variables in the code above are not assigned with an affine transformation of the thread index (in integer arithmetic). Lets call Kim's systems handling affine values *CAE* systems (compact affine execution) and call our systems handling the code above *CBE* systems, with the B standing for bunched. A CBE system uses a *BSRF*, which is the equivalent of the ASRF in the CAE systems. Our goal is that the CBE system can efficiently handle all cases that CAE systems can handle, plus the modulo and integer division cases above.

The good news for CBE systems is that they can handle code that bunches threads, such as the code fragment above. The bad news is that whereas CAE systems could add any two ASRF values and write the result into the ASRF, the CBE systems can only do that for certain pairs, for other cases the result will have to be written into the regular register file. (See Section 2.3. Let $V_{\text{h0}}(i)$ $V_{\text{b0}}(i)$ denote the compact representation of `h0` and `b0` above. If the division in the expressions for `h0` and `b0` were replaced by multiplication there would be no problem with a compact representation of `hc=h0+b0`, $V_{\text{hc}}(i) = i \times (N+C)$. When such a value is stored in the ASRF the stride field holds a single value, the sum $N+C$. This doesn't work for integer division because a single value obtained by evaluating $\frac{1}{N} + \frac{1}{C}$ could not be used to compute the same value of $\lfloor i/N \rfloor + \lfloor i/C \rfloor$.)

(*a*) Re-write Section 2.3, Value Structure, of Kim 2013 for our CBE system. Be sure to include a replacement for $V(i) = b + i \times s$. Also modify the section describing addition and multiplication of the compact values to account for the moduli and division. Describe when it is and is not possible to represent the result of such arithmetic on compact values.

(*b*) Describe what kind of information the BSRF will store. Also describe how the stride functional unit (see Figure 4) will have to be modified.

(*c*) Describe how the equivalent of affine memory operations (see page 135) will be different in CBE than in CAE. Describe the difference in performance and hardware between the two.

Problem 3: Consider the loop used for loading shared memory in the `mxm_volk` kernel:

```
auto idx = [](int i,int r,int c) { return i * N*N + r * N + c; };
const int tid = threadIdx.x + blockIdx.x * blockDim.x;
const int start = tid / N;
const int r0 = ( threadIdx.x % N ) / CS;
const int c0 = threadIdx.x % CS;

for ( int h=start; h<stop; h += inc )
    for ( int cc=0; cc<N; cc += CS )
        for ( int rr = 0; rr<N; rr += RS )
          SHARED_MEM = d_app.d_a[ idx( h, rr + r0, cc + c0 ) ];
```

Our goal is to use one BSRF value for the address `&d_app.d_a[ idx( h, rr + r0, cc + c0 ) ];`. With ASRF that would be easy for `&d_app.d_a[tid]`. But even with BSRF it would be a challenge because we would need one compact representation of something computed from three compact representations, `h`, `r0`, and `c0`, plus the constant values (which are easy) `&d_app.d_a`, `rr`, and `cc`. As mentioned in an earlier problem one can't find a compact representation of the sum of compact representations that include division and moduli.

Suppose that `N` and `CS` are both powers of 2. Then expressions for `r0` and `c0` in the code above can be viewed as rearranging the bits in the binary representation of `threadIdx.x`. Lets call a system based on this idea *CSE*, for compact swizzled execution. A compact value is represented by a base plus a swizzling of bits in the thread index. For example, consider `x = a + threadIdx.x * 4`. The base would be `a` and the offset would be $i_9 i_8 \cdots i_1 i_0 00$, where i_p is the value of bit position p in the binary representation of the thread index. The original value of the thread index is $i_9 i_8 \cdots i_1 i_0$. A representation of `x = a + threadIdx.x % 8` would be $a + i_2 i_1 i_0$. This swizzling of the bits of the thread index can be represented by a ten-element array π, where bit p of the thread index is placed in position $\pi(p)$ in the swizzled value (with, say, $\pi(p) = \emptyset$ indicating that bit p should not appear in the swizzled value).

(a) Show how values compactly represented this way can be added under certain circumstances. That is, two bases and two π's will be added. Note that it is not always possible to add two such values. An example of what is possible is adding the representations of `x=(threadIdx.x/64)` and `y=(threadIdx.x%16)*64` to obtain `z=x+y`.

(b) Show how the memory address needed in the code above can be represented with a single CSE CSRF register. Assume powers of two for CS and N.

(c) Compare the cost of the CSRF to the BSRF and the ASRF.

```
extern "C" __global__ void mxm_volk(Elt_Type* __restrict__ dout)
{
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int num_threads = blockDim.x * gridDim.x;

  // Convenience function for finding index of the element at row r,
  // column c, in matrix i.
  //
  auto idx = [](int i,int r,int c) { return i * N*N + r * N + c; };

  const int start = tid / N;
  const int stop = d_app.n_mats;
  const int inc = num_threads / N;

  // Chunk Size: Number of columns of matrix A to load into shared memory.
  //
  const int CS = 32 / sizeof(Elt_Type);

  // Column in matrix B assigned to this thread.
  //
  const int cb = threadIdx.x % N;

  // Column offset to load when populating shared memory.
  //
  const int c0 = threadIdx.x % CS;

  // First row to load when populating shared memory.
  //
  const int r0 = ( threadIdx.x % N ) / CS;

  const int h0 = threadIdx.x / N;

  // Number of times per matrix that shared memory will have to be loaded.
  //
  const int RS = N / CS;
  const int MpB = 32 * 32 / N;    // Matrices per block.

  // Storage for buffering N by CS submatrix of matrix A.
  //
  __shared__ Elt_Type mat_a[N][MpB][CS];

  for ( int h=start; h<stop; h += inc )
    {
      // Storage for column of output matrix.
      //
      Elt_Type elt[N];
      for ( auto& e: elt ) e = 0;

      for ( int cc=0; cc<N; cc += CS )
        {
          // Write shared memory with an N by CS submatrix of A.
          //
          for ( int rr = 0; rr<N; rr += RS )
            mat_a[rr + r0][h0][c0] =
              d_app.d_a[ idx( h, rr + r0, cc + c0 ) ];

          if ( N > 32 ) __syncthreads();

          for ( int rb=0; rb<CS; rb++ )
            {
              const int r = cc + rb;  // Row in matrix B, column in mat A.
              Elt_Type elt_rb_cb = d_app.d_b[ idx( h, r, cb ) ];
              for ( int ra=0; ra<N; ra++ )
                elt[ra] += mat_a[ra][h0][rb] * elt_rb_cb;
            }

          if ( N > 32 ) __syncthreads();

        }
      for ( int r=0; r<N; r++ )
        dout[ idx( h, r, cb ) ] = elt[r];
    }
}
```