Sparse Matrix-Vector Multiplication
for the ClearSpeed Accelerator

Anna Youssefi
December 7, 2007

1 Introduction
Sparse matrix-vector multiplication (SpMV), \( y = A \times x \), where A is a sparse matrix and x, y are vectors, is a common computational kernel in many application domains that presents challenges for performance optimization. The high ratio of memory operations to computation and the lack of data reuse cause sparse matrix-vector multiplication to be bandwidth intensive. Additionally, the application areas for this kernel usually require double precision floating point support. These factors have prevented accelerators from delivering any speedup over single-CPU performance for this kernel; indeed, performance on accelerators has typically been lower than optimized performance on single-node CPUs. [TODO: include citations] Recent work by Williams et al. [2] has achieved speedups from parallel SpMV on multicore processors as well as the heterogeneous Cell.

In this paper, we examine hand-tuning SpMV for the ClearSpeed accelerator. Unlike traditional GPUs, ClearSpeed provides double precision floating point support, making it a better candidate for off-loading general-purpose scientific computation. By coupling a well-suited data representation, Length-grouped Compressed Sparse Row (L-CSR) format, with careful orchestration of memory operations, we hope to achieve parallel speedup over single node CPU performance on the ClearSpeed.

In Section 2, we summarize some of the relevant features of the ClearSpeed architecture, runtime environment, and programming API. In Section 3 we discuss the L-CSR data representation and compare it with common alternatives. In Section 4 we introduce a naive implementation of sparse matrix-vector multiplication for the ClearSpeed accelerator, and in section 5 we attempt to improve performance over the naive version.

2 The ClearSpeed Accelerator
The ClearSpeed Advance Accelerator board has two CSX600 processors and 1 GB of DDR2 SDRAM. Each processor can access one half of the SDRAM on the board. The CSX600 processor consists of mono and poly execution units. The mono unit handles instruction fetch and dispatch, program control flow, and thread scheduling. It has an execution unit, a semaphore unit, a 32 x 16-bit register file, an 8 KB instruction cache, and a 4 KB data cache. There is also a 128 KB scratchpad memory. The hardware supports 8 threads with software-controlled thread switching.

The poly execution unit is a SIMD array of 96 processing elements, each with a 128-byte register file and 6 KB of local memory. Data must be explicitly transferred from mono to poly memory, and
both synchronous and asynchronous memory operations are provided by the API. Asynchronous
data transfers rely on hardware-implemented semaphores for synchronization.

Semaphores can also be used to coordinate between the host and the mono execution unit, and
both synchronous and asynchronous memory operations are also available to transfer data from
the host to mono memory. However, presently only the synchronous host-to-board data transfers
provide alternatives that don’t halt the ClearSpeed processors during data transfer.

Theoretical peak bandwidth is 96 GB/s for internal (on chip) memory and 3.2 GB/s for exter-
nal (on board) memory. The vendor states over 25 GFLOPS can be sustained for dense matrix
multiplication (DGEMM). 1

The ClearSpeed Accelerator provides the C^n language as the programming interface for the
CSX600 processors. C^n is an extension of C that introduces features such as the mono and poly
multiplicity specifiers and some control logic modifications to support programming for the CSX
architecture. Some standard libraries are provided, including memory and print operations, in
addition to specialized math libraries that exploit the SIMD functionality of the accelerator.

The ClearSpeed Application Programming Interface (CSAPI) enables the host program to com-
municate with code running on the accelerator. This includes data transfers between the host and
mono memory, synchronization between the host and mono execution unit using semaphores, and
the API to initiate and run the code on the mono unit.

3 Data Representation

A common data representation for sparse matrices is compressed sparse row format (CSR), in which
the non-zero elements are stored in one contiguous data vector and additional data vectors are used
for storing the starting position and column indices for each row. This format has the benefit of
reducing unnecessary storage, loads, and computation for zeros but at the cost of additional loads
for the column indices.

When performing sparse-matrix vector multiply with sparse matrices in which many rows share
a short common length, prior work by Mellor-Crummey and Garvin [1] has shown benefits for
both instruction mix and schedule from a different permutation of the CSR format, length-grouped
compressed sparse row. L-CSR format regroups rows by length (number of non-zero entries). An
additional indirection couples the lengths with indices of the starting rows for each length.

CSR format prevents effective software pipelining since the computation of each row involves a
recurrence. Grouping same-sized rows together enables applying unroll-and-jam to interleave the
computation of different rows of the same length. Assuming the absence of memory stalls, this fills
the floating point pipeline with multiple independent recurrences.

For common row lengths, fully-unrolled special case code can be inserted, completely eliminating
branch overhead for rows of those lengths and enabling the compiler to significantly reduce the
address arithmetic. Additionally, L-CSR can facilitate SIMD execution because rows of the same
length can be handled uniformly.

Another approach is register blocking, which groups nonzeros into rectangular tiles and requires
only a single column index per tile. Tiles may be fully dense, as in block compressed sparse row
(BCSR), or may contain zero elements explicitly to enable larger-sized blocks, thereby reducing
the number of total indices needed. To accommodate the SIMD units and stream buffers on the

1 Performance data in the CSX600 Product Brief is listed differently than in the Advance X620 and e620 Product
Briefs. The former is used here.
Cell architecture, Williams et al. use block coordinate (BCOO) format, storing zeros explicitly and requiring a row and a column index, but no row pointers, for each block [2].

4 A Naive L-CSR Implementation for ClearSpeed

In this section we introduce a naïve implementation of SpMV for the ClearSpeed processor. Single-CPU code written in the C language is provided in Appendix A for comparison.

We start with two simplifications that will undoubtedly have a dire impact on performance: first, all memory transfers between the host processor and board are completed before and after all of the computation; and second, we use synchronous data transfers between mono and poly memory. Therefore, this implementation provides us with a lower bound for performance. Additionally, it allows us to measure the amount of time spent performing data transfers versus computation. Quantifying these separately will help us determine whether any speedup relative to single CPU performance might be gained from overlapping data transfers and computation.

We make one performance-based choice in this version. We leave the results in the L-CSR permuted row order and reorder them back to original row ordering on the host after they have been transferred back. This helps performance by allowing the PEs to write back (to mono memory) contiguous blocks of data for the results. Additionally, leaving the results in permuted order eliminates the need to transfer row indices to the PEs, or even to the accelerator if the host handles reordering the results.

The C code for the computation kernel is provided in Appendix B. We do not show the CSAPI code for this implementation since the host and accelerator do not communicate during the computation. It is assumed that all mono memory has been allocated and all data received before the computation begins and that the results are sent back to the host after the computation has ended. Once the host has received the results, it must permute them back to the original row order.

When running this version, we want to measure how much time is spent transferring data between the host and accelerator. The total time measurement will include this data transfer as well as the results reordering on the host. Since we use synchronous data transfers from mono to poly memory, we can also measure how much time we spend on such transfers by signaling the host before and after each transfer and measuring the time in between on the host.

5 Improving Performance

In order to hand-tune our code to achieve better performance on this architecture, we suggest four modifications to the naïve implementation: improvements for data transfer between mono and poly memory, applying unroll and jam to reduce pipeline stalls, the use of vector instructions, and better host coordination and partitioning of the computation.

5.1 Data Transfer Improvements

The naïve implementation uses synchronous copying functions to transfer data between mono and poly memory. The most important modification is to use the asynchronous routines with double buffering to overlap data transfers and computation to the extent possible.

Additionally, the memory transfer operations between mono and poly memory stipulate that for optimal performance, the mono data should be 32-byte aligned. If the rows are blocked in a
multiple of 4, this alignment can be maintained for the nonzeros and results. However, the case in which PEs compute only a single row instead of a block of rows would need to be eliminated in favor of only computing blocks of rows on some of the PEs and leaving others idle. Other approaches for maintaining an optimal alignment are also possible in the event that the performance benefits outweigh the costs.

5.2 Unroll-and-Jam

Applying unroll-and-jam to interleave the computation of multiple rows on each PE might help reduce pipeline stalls but requires that all x vector values for the rows be fetched prior to the start of the computation (unlike in the naive implementation, in which we fetch one row at a time). Furthermore, it enables the use of vector instructions.

5.3 Vector Instructions

The CSX600 architecture includes a set of hardware instructions that “allow the programmer to operate on groups of operands as a block and thus utilize the pipelined nature of the units within the poly processor. These instructions can also use more than one functional unit within the poly processor in a single instruction, again increasing the amount of work done per cycle.” (Section 4.11, CSX600 Hardware Programming Manual) Vector intrinsic functions are provided for programming these hardware instructions (presented in ClearSpeed SDK Reference Manual, Section 12.12).

By using a multiple of 4 row block size and applying unroll-and-jam as mentioned in the previous subsection, we can exploit the vector functionality of the PEs to compute 4 rows at a time in a pipelined fashion. The vector multiply accumulate intrinsic function “multiplies the elements of vector x by the corresponding elements of vector y and adds the result to vector z, returning z.” We can use this function to multiply one nonzero by one value of the x vector and accumulate into each result for 4 rows.

As an alternative for when we compute a single row at a time, we can use a vector reduction for 4 elements of the same row. The vector multiply reduction “multiplies the elements of vector x by the corresponding elements of vector y then adds the results to produce a scalar result.” We can simply block the computation of the row by 4 nonzeros and finish any remaining nonzeros with regular instructions.

Prototypes for both of these intrinsics are shown in Figure 1.

```c
__DVECTOR __cs_vmulacc(__DVECTOR z, __DVECTOR x, __DVECTOR y);
poly double __cs_vreduce_mul(__DVECTOR x, __DVECTOR y);
```

Figure 1: Vector Intrinsic Function Prototypes

5.4 Host Coordination

In our naive implementation, all data is transferred from the host to one CSX600 processor on the accelerator board, the computation is performed on that processor, and then all the results are transferred back to and reordered on the host. It should be possible to improve performance by
overlapping the communication of data from the host with computation on the accelerator, using semaphores and double buffering. Furthermore, we would like to use both CSX600 processors on the accelerator board instead of just one. Unfortunately, the two accelerator processors do not share memory. Therefore, the host will have to send both of them copies of the x vector if the computation is partitioned among them.

6 Conclusion

While we obviously won’t know how our naive implementation or suggested improvements perform until we can run code on the ClearSpeed Accelerator, the ability to achieve a parallel speedup will undoubtedly depend on how well we are able to tolerate the data transfer latencies, both between host and board and between mono and poly memory. Of particular concern are the mono to poly copies from the x vector, since these are not contiguous. While the architecture can support these accesses, we don’t know what the performance penalties will be. (We are currently even unaware of the latency for on-board external memory accesses.) However, if we can effectively prefetch these using asynchronous memory operations, the SIMD parallelism enabled by the L-CSR format might be able to exploit enough of the architectural features to yield acceleration on this bandwidth-bound computation kernel.

References


A Single-CPU L-CSR SpMV code

#define GROUP_START 1

void
lcsr_mvmult(int lcsr_numlengths, int *lcsr_lengths, int *lcsr_rowindex,
           int *lcsr_cols, double *lcsr_nz, double *x, double *r)
{
   int col, len, row, rowlen;
   int firstrow, lastrow, i;
   double result;

   col = 0;

   for(len = 0; len < lcsr_numlengths * 2; len += 2){
      firstrow = lcsr_lengths[GROUP_START + len];
      rowlen = lcsr_lengths[len];
      lastrow = lcsr_lengths[GROUP_START + len + 2] - 1;

      for (row = firstrow; row <= lastrow; row += 1){
         //compute all rows of same length
         result = 0.0;
         for (i = 0; i < rowlen; i++){
            //compute one row
            result = result + lcsr_nz[col] * x[lcsr_cols[col]];
            col = col + 1;
         }
         r[lcsr_rowindex[row]] = result;
      }
   }

   return;
}
#include <string_ext.h>

#define GROUP_START 1

#define NUM_PES 96 // number of SIMD Processing Elements
#define ROW_BLOCK // number of rows each PE will receive and compute at a time

/* this is a property of the matrix structure
* either #define or pass it as a runtime parameter
*/
#define MAX_ROW_LEN the longest row length for the sparse matrix

/* Listed here just for reference; used by host to allocate mono memory */
#define TOTAL_NZ number of nonzero elements in matrix
#define TOTAL_ROWS number of total rows to be computed

/*****************************************************/
/* Mono Memory w/Shared Addresses with Host */
// Memory allocated by host using CSAPI

/* HOST-WRITE: */
// Data transferred from host, total size of data shown in parens
mono short lcsr_numlengths;
mono int * lcsr_lengths;
mono double * mono nonzeros; // (TOTAL_NZ)
mono int * mono colindices; // (TOTAL_NZ)
mono double * mono xvector; // (TOTAL_ROWS)

/* HOST-READ: */
// Points to data to be transferred to host, total size of data shown in parens
mono double * mono permutedresults; (TOTAL_ROWS)

/*****************************************************/
/* POLY BUFFERS: */
poly double pe_nonzeros[ROW_BLOCK * MAX_ROW_LEN];
poly int pe_colindices[ROW_BLOCK * MAX_ROW_LEN];
poly double pe_xvect[ROW_BLOCK * MAX_ROW_LEN];
poly double tempresults[ROW_BLOCK];

mono int col, row, numrowsleft;
mono short len, rowlen, i, j, k;

/* These pointers are only needed if the strided memory copying functions aren’t available
* for synchronous transfers or can’t be used due to alignment. The documentation

claims strided operations are available for both but they aren’t listed in the
Standard Library Reference for the synchronous functions.

```c
mono double * poly nzindex;
mono int * poly colindex;

mono double * poly xval;
mono double r;
poly double res;
poly short penum = get_penum(); //may not be needed
unsigned short semnotused 0;
col = 0;

for(len = 0; len < lcsr_numlengths * 2; len += 2){
  row = lcsr_lengths[GROUP_START + len];
  rowlen = lcsr_lengths[len];
  lastrow = lcsr_lengths[GROUP_START + len + 2] - 1;

  while (lastrow >= row) {
    numrowsleft = lastrow - row + 1;

    if (numrowsleft >= ROW_BLOCK * NUM_PES) { // Each pe computes ROW_BLOCK rows

      // if we use strided ops we don’t need to compute these pointers!
      // col would be starting location, stride would be ROW_BLOCK * rowlen,
      // participation number = penum
      nzindex = nonzeros + col + (penum * ROW_BLOCK * rowlen);
      colindex = colindices + col + (penum * ROW_BLOCK * rowlen);

      memcpym2p(pe_nonzeros, nzindex, ROW_BLOCK * rowlen * 8);
      memcpym2p(pe_colindices, colindex, ROW_BLOCK * rowlen * 4);

      k=0;
      for (i=0; i < ROW_BLOCK; i++){
        // fetch xvector values for 1 row at a time
        for (j=k; j < k + rowlen; j++) {
          xval = xvector + pe_colindices[j];
          memcpym2p(pe_xvect[j], xval, 8);
        }
        k=j;
        res = 0.0;
        for (j = k - rowlen; j < k; j++){
          res += pe_nonzeros[j] * pe_xvect[j];
        }
        tempresults[i] = res;
      }
    }
```

8
// ROW_BLOCK doubles written per PE starting at permutedresults[row]
mempcpy2m_strided(permutedresults + row, tempresults, 8 * ROW_BLOCK);

col += NUM_PES * ROW_BLOCK * rowlen;
row += NUM_PES * ROW_BLOCK;

}  
else if (numrowsleft >= NUM_PES) {  // Each pe computes 1 row
    nzindex = nonzeros + col + (penum * rowlen);
    colindex = colindices + col + (penum * rowlen);

    memcpym2p(pe_nonzeros, nzindex, rowlen * 8);
    memcpym2p(pe_colindices, colindex, rowlen * 4);

    // fetch xvector values for 1 row
    for (j=0; j < rowlen; j ++) {
        xval = xvector + pe_colindices[j];
        memcpym2p(pe_xvect[j], xval, 8);
    }
    res = 0.0;
    for (j =0; j < rowlen; j++){
        res *= pe_nonzeros[j] * pe_xvect[j];
    }
    tempresults[0]= res;  //can we write res back directly instead?

    // 1 double written per PE starting at permutedresults[row]
mempcpp2m_strided(permutedresults + row, &res, 8);
    col += NUM_PES * rowlen;
    row += NUM_PES;

}  
else {  //finish it up on mono unit to avoid data transfer
    for (; row <= lastrow; row ++){
        r = 0.0;
        for (j =0; j < rowlen; j++){
            r += nonzeros[col] * xvector[colindices[col]];
            col ++;
        }
        permutedresults[row] = r;
    }
}

// write back results to host and reorder results on host