B629 project - StreamIt MPI Backend

Nilesh Mahajan

March 26, 2013
Abstract

StreamIt is a language based on the dataflow model of computation. StreamIt consists of computation units called filters connected with pipelines, split-joins, feedback loops. Thus, the program is a graph with filters as vertices and the connections as edges. The compiler then estimates the workload statically and computes a partitioning of the graph and assigns work to threads. StreamIt also has a cluster backend that generates threads which communicate through sockets. MPI is a popular library used in the HPC community to write distributed programs written in the Single Program Multiple Data (SPMD) style of programming. MPI community over the years has done a lot of work to identify, optimize communication patterns in various HPC applications. The idea of this project was to see if there are any opportunities to identify collective communication patterns in StreamIt programs and generate efficient MPI code accordingly.
1 Introduction

StreamIt[4] is a language developed at MIT to simplify writing streaming applications in domains such as signal-processing. It is based on dataflow model of computation where the program consists of basic computation units called filters. The language the filters are written in looks like Java. Filters are connected with structures/channels like pipelines, split-joins, feedback loops to form a graph. The filters can have state although it is discouraged. The channels have known data rates which specify the number of elements moving through them. The resulting graph is a structured graph which is easier for compiler to optimize still general enough so that the programmer can write useful programs[5]. This kind of dataflow model is generally referred to as synchronous dataflow. StreamIt adds features like dynamic data rates, sliding window operations, teleport messaging. Mainly three kinds of parallelism are exploited: task (fork-join parallelism), data (DOALL loops), pipeline (like ILP in hardware)[1]. The compiler does a static estimation of work and applies various transformations like fusion, fission of filters, cache-aware optimizations to the stream graph. The StreamIt program is translated to Java code which can be run with other Java library or C code. There is also a cluster backend that generates code that communicates via sockets.

2 MPI Model

MPI[2] is a library used in the HPC community to write distributes program in the SPMD model. There are $n$ processes each with a unique id called rank ranging from 0 to $n$. Each process operates in a separate address space and communicates data through primitives provided by MPI. There are primitives for point-to-point communication (MPI_Send, MPI_Recv), collectives (MPI_Broadcast, MPI_Reduce etc.), one-sided communication. The MPI community has been actively working on identifying and optimizing collective patterns common across various HPC domains[3]. Also vendors provide their own implementations of collectives taking advantage of specialized hardware. For example, implementing a broadcast as a for loop with point to point send, receives is almost always slower than MPI_Broadcast even on shared memory. So the idea of this project was to see if we can identify collective patterns in stream programs and generate MPI code instead of the socket based cluster backend.

3 StreamIt Install

In this section, I discuss the difficulties faced installing StreamIt on my laptop. My system is Ubuntu 12.10, dual-core Intel x86-64. I downloaded source from the git repo and tried to install it according to the installation instructions in the source code. One of the problems faced was that you need to add the following paths to CLASSPATH:
Then I needed to add the following line

```java
p . print ( "#include <limits.h>\n" );
```

at line 44 in `src/at/dms/kjc/cluster/GenerateMasterDotCpp.java`. I think with the newer gcc (≥ 4.4) the generated code needs `limits.h` to compile successfully. After this I was able to successfully compile the StreamIt programs.

## 4 Benchmarks

We evaluate the performance of two typical linear algebra kernels written in StreamIt versus C MPI, namely Matrix Multiplication, Cholesky Factorization. First we compare the programs written in C with MPI versus the corresponding StreamIt versions because they look significantly different.

### 4.1 Matrix Multiply

The program starts its execution in the MatrixMult method which is the name of the file. This is similar to Java where the name of the public class is equal to the name of the file. There are three filters in the pipeline namely FloatSource, MatrixMultiply, FloatPrinter. FloatSource is a stateful filter that produces numbers from 1 to 10. The size of the matrices is 10x10. Then there are filters to transpose the second matrix and multiply, accumulate the result in parallel. One can compile the code with the `strc` command. Figure 1 denotes the graph generated from the program. I was not sure how to pass the size argument as a command line option to the program and could not find anything in the source or documentation. So I generated various versions of the program with different matrix sizes. The compiler does a lot of work in generating the optimal version so it takes lot of time to compile programs. It is especially interesting that the compiler takes more time to compile programs with higher matrix sizes and even fails for large matrix sizes. Here are the times the compiler took to
compile programs with different sizes:

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>real time</th>
<th>user time</th>
<th>sys time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0m3.628s</td>
<td>0m4.116s</td>
<td>0m0.228s</td>
</tr>
<tr>
<td>100</td>
<td>0m5.445s</td>
<td>0m7.812s</td>
<td>0m0.292s</td>
</tr>
<tr>
<td>200</td>
<td>0m13.419s</td>
<td>0m12.145s</td>
<td>0m0.652s</td>
</tr>
<tr>
<td>300</td>
<td>0m39.196s</td>
<td>0m25.250s</td>
<td>0m1.272s</td>
</tr>
</tbody>
</table>

Figure 1: StreamIt Matrix Multiply Graph

The compilation failed for sizes above 500 elements. For example, for matrix size 1000, compilation failed with the following error after taking 5m14.521s of real time:

combined_threads.o: In function ‘init_FloatSource__32003_81__0()’:

We can pass the number of iterations to the executable with −i command line option. Following is the StreamIt code:

```c
void->void pipeline MatrixMult {
    add FloatSource(10);
    add MatrixMultiply(10, 10, 10, 10);
    add FloatPrinter();
}

float->float pipeline MatrixMultiply(int x0, int y0, int x1, int y1) {
    // rearrange and duplicate the matrices as necessary:
    add RearrangeDuplicateBoth(x0, y0, x1, y1);
    add MultiplyAccumulateParallel(x0, x0);
}
```
float->float splitjoin RearrangeDuplicateBoth(int x0, int y0, int x1, int y1) {
    split roundrobin(x0 * y0, x1 * y1);
    // the first matrix just needs to get duplicated
    add DuplicateRows(x1, x0);
    // the second matrix needs to be transposed first
    // and then duplicated:
    add RearrangeDuplicate(x0, y0, x1, y1);
    join roundrobin;
}

float->float pipeline RearrangeDuplicate(int x0, int y0, int x1, int y1) {
    add Transpose(x1, y1);
    add DuplicateRows(y0, x1*y1);
}

float->float splitjoin Transpose(int x, int y) {
    split roundrobin;
    for (int i = 0; i < x; i++) add Identity<float>();
    join roundrobin(y);
}

float->float splitjoin MultiplyAccumulateParallel(int x, int n) {
    split roundrobin(x*2);
    for (int i = 0; i < n; i++) add MultiplyAccumulate(x);
    join roundrobin(1);
}

float->float filter MultiplyAccumulate(int rowLength) {
    work pop rowLength*2 push 1 {
        float result = 0;
        for (int x = 0; x < rowLength; x++) {
            result += (peek(0) * peek(1));
            pop();
            pop();
        }
    push(result);
}

float->float pipeline DuplicateRows(int x, int y) {
    add DuplicateRowsInternal(x, y);
}
float->float splitjoin DuplicateRowsInternal(int times, int length) {
    split duplicate;
    for (int i = 0; i < times; i++) add Identity<float>();
    join roundrobin(length);
}

void->float stateful filter FloatSource(float maxNum) {
    float num;
    init {
        num = 0;
    }
    work push 1 {
        push(num);
        num++;
        if (num == maxNum) num = 0;
    }
}

float->void filter FloatPrinter {
    work pop 1 {
        println(pop());
    }
}

float->void filter Sink {
    int x;
    init {
        x = 0;
    }
    work pop 1 {
        pop();
        x++;
        if (x == 100) {
            println("done..");
            x = 0;
        }
    }
}

This is how the C sequential version of matrix multiply looks like:

    */* initialize A */*
    for(j=0; j < rows1; j++) {
        for(i=0; i < columns1; i++) {
            A[i][j] = floatSource();
        }
    }
** initialize B **/
for(j=0; j < rows2; j++) {
    for(i=0; i < columns2; i++) {
        B[i][j] = floatSource();
    }
}

for(j = 0; j < rows1; j++) {
    for(i = 0; i < columns2; i++) {
        int k;
        float val = 0;

        /** calculate C_{i,j} **/
        for(k = 0; k < columns1; k++) {
            val += A[k][j] * B[i][k];
        }
        C[i][j] = val;
    }
}

The initialization loops populate the A and B matrices through values generated from a method called floatSource which is equivalent to the StreamIt filter. I was unable to run the generated cluster code on an actual cluster like odin because there were some libraries missing on odin. Another problem is I am unable to compile StreamIt code with larger matrix sizes hence the MPI comparison might not be a fair comparison because MPI has advantages when it comes to larger array sizes. Still I compared the sequential, StreamIt and MPI codes with the results shown in Figure 2. The x-axis is array sizes, the y-axis is time in nanoseconds.

As can be seen in the figure, the MPI version performs worse than both the StreamIt(strc) and sequential versions. This can be expected since MPI has copying overheads which will dominate computation times. So we need higher array sizes for a fair comparison. We can also see that the sequential version is better than the StreamIt version. My guess is again the StreamIt code suffers from communication overheads.

### 4.2 Cholesky Factorization

The second kernel we look at is Cholesky Factorization. Again it took a lot of time compiling the StreamIt source and it took longer to compile programs with higher array sizes. Following are the times:

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>real time</th>
<th>user time</th>
<th>sys time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1m53.304s</td>
<td>1m45.375s</td>
<td>0m1.792s</td>
</tr>
<tr>
<td>150</td>
<td>3m26.589s</td>
<td>3m11.444s</td>
<td>0m2.184s</td>
</tr>
<tr>
<td>200</td>
<td>5m0.306s</td>
<td>3m57.171s</td>
<td>0m3.284s</td>
</tr>
</tbody>
</table>


Figure 2: Matrix Multiply Comparison

For size 300, the compilation took around 45 minutes and terminated with a StackOverflow exception. Figure 3. The x-axis is array sizes, the y-axis is time in nanoseconds. I tried the MPI versions with different number of processors (4, 9, 16). The MPI always performs worse while the sequential performs better. My guess is again this is due to communication overheads dominating actual computation.

Here is the C sequential kernel of Cholesky Factorization.

```c
// cholesky factorization
for (k=0; k < N; k++) {
    A_(k, k) = sqrt(A_(k, k));
    for (i=k+1; i < N; i++)
        A_(i, k) /= A_(k, k);
    for (i=k+1; i < N; i++)
        A_(i, k) /= A_(k, k);
}```
for \( j = k + 1; j \leq i; j++ \)
\[
A_{i,j} = A_{i,k} \ast A_{j,k};
\]

Here is the StreamIt version of Cholesky Factorization.

```c
void->void pipeline chol() {
    int N=100;
    add source(N);
    add rchol(N);
    // this will generate the original matrix, for testing purposes only
    add recon(N);
    add sink(N);
}
```
/*
 * performs the cholesky decomposition
 * on an N by N matrix where input elements
 * are arranged column by column with no redundancy,
 * such that there are only N(N+1)/2 elements
 * This is a recursive implementation
 */

float->float pipeline rchol(int N) {
    // this filter does the divisions
    // corresponding to the first column
    add divides(N);
    // this filter performs the updates
    // corresponding to the first column
    add updates(N);
    // this splitjoin takes apart
    // the first column and performs the chol(N-1)
    // on the resulting matrix it is
    // where the actual recursion occurs.
    if (N>1)
        add break1(N);
}

float->float splitjoin break1(int N) {
    split roundrobin(N,(N*(N-1))/2);
    add Identity<float>();
    add rchol(N-1);
    join roundrobin(N,(N*(N-1))/2);
}

// performs the first column divisions

float->float filter divides(int N) {
    init {}

    work push ((N*(N+1))/2 pop (N*(N+1))/2 {
        float temp1;
        temp1=pop();
        temp1= sqrt(temp1);
        push(temp1);

        for (int i=1; i<N; i++)
            push(pop()/temp1);

        for (int i=0; i < (N*(N-1))/2; i++)
            push(pop());
    }
}
push(pop());
}
}

// updates the rest of the structure
float->float filter updates(int N) {

  init()

  work pop (N*(N+1))/2 push (N*(N+1))/2 {
    float [N] temp;

    for (int i=0; i<N; i++)
      temp[i]=pop();
    push(temp[i]);

    for (int i=1; i<N; i++)
      for (int j=i; j<N; j++)
        push(pop()-temp[i]*temp[j]);
  }
}

// this is a source for generating
// a positive definite matrix
void->float filter source(int N) {

  init()

  work pop 0 push (N*(N+1))/2 {
    for (int i=0; i<N; i++)
      push((i+1)*(i+1)*100);
    for (int j=i+1; j<N; j++)
      push(i*j);
  }
}

// prints the results:
float->float filter recon(int N) {

  init()

  work pop (N*(N+1))/2 push (N*(N+1))/2 {
    float [N][N] L;
    float sum=0;

for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        L[i][j]=0;

for (int i=0; i<N; i++)
    for (int j=i; j<N; j++)
        L[j][i]=pop();

for (int i=0; i<N; i++)
    for (int j=i; j<N; j++){
        sum=0;
        for (int k=0; k<N; k++)
            sum+=L[i][k]*L[j][k];
        push(sum);
    }
}

5 Cluster Backend

The \texttt{-clusterN} option selects a backend that compiles to \(N\) parallel threads that communicate using sockets. When targeting a cluster of workstations, the sockets communicate over the network using TCP/IP. When targeting a multi-core architecture, the sockets provide an interface to shared memory. A hybrid setup is also possible, in which there are multiple machines and multiple threads per machine; some threads communicate via memory, while others communicate over the network. In order to compile for a cluster of workstations, one should create a list of available machines and store it in the following location:
STREAMIT_HOME/cluster-machines.txt. This file should contain one machine name (or IP address) per line. When the compiler generates N threads, it will assign one thread per machine (for the first N machines in the file). If there are fewer than N machines available, it will distribute the threads across the machines.

The compiler generates cpp files named threadsN.cpp where N ranges from 0 to a number the compiler determines based on the optimizations. For example for the matrix multiply program with matrix size 10, 13 cpp files were generated while for cholesky with the same matrix size, 500 cpp files were generated contributing to the enormous compile time. Finally, a combinedNthreads.cpp connects everything together. The cluster code is generated by compiler files in the directory STREAMIT_HOME/src/at/dms/kjc/cluster. The file GenerateMasterDotCpp.java generates the combinedNthreads.cpp file while the socket code is generated with ClusterCodeGenerator.java, FusionCode.java and FlatIRToCluster.java files. To generate MPI code, we will need to modify these files.

6 Conclusion

Although no definitive conclusion can be drawn from this study as for the usability of writing an MPI backend for StreamIt, I got to explore and understand the StreamIt model of computation better. Given more time and a better study with different benchmarks, we should be able to determine whether it is beneficial to generate MPI code instead of the socket code generated right now. It looks likely that the MPI backend should be beneficial by just looking at the cpp code generated for Cholesky, Matrix Multiply. It will take another term project to identify the communication pattern statically from the StreamIt source and modify parts of the compiler to generate MPI code.
Bibliography


