1 Introduction

In CS611 we only considered a single level of tiling. A natural question is what could one do for a memory hierarchy of greater depth? Some people argue that the divide-and-conquer approach to writing programs is the right one, as each successive subproblem operates on a smaller data set of its own, therefore improving data locality.

In this project your goal is to generate BLAS codes optimized for locality for the different levels of the memory hierarchy by starting with a block-recursive description. You will have to use the newest version of BCCK (a.k.a. ISSFX.CCK) to design and implement source-to-source transformations from the BRILA (Block-Recursive Implementation of Linear Algebra kernels) language to C. Then you are going to apply restructuring transformations to the resulting code in order to achieve near-optimal performance. Although the project is already underway, it is in its initial stages of design/development and you can make a lot of impact on the final outlook.

The general steps of the process are as follows.

1. **Produce a High-Level Description of the Problem** – Use a domain-specific language to describe dense linear algebra problems in a block-recursive fashion.

2. **Generate an Iterative Version** – Automatically convert the provided domain-specific, block-recursive description to a loop nest.

3. **Apply Restructuring Optimizations** – Apply well known restructuring transformations, defining new optimization parameters to drive them if needed.

4. **Determine Optimization Parameters** – Use empirical and modelling techniques to determine appropriate values for the optimization parameters in the iterative version.

5. **Generate Library Code** – Use the back-end low-level compiler to produce object files to be included in the library.

These steps are described in below in more detail.

2 Produce a High-Level Description of the Problem

The problem is described using a high-level domain-specific language (DSL) in order to preserve as much semantic information as possible. It is well known that when an algorithm is expressed in a given language, depending on the abstraction level of the chosen language, some of the semantic information is lost and some of the possible implementation choices are no longer available.

\[
\begin{array}{cc}
A_{11} & A_{12} \\
A_{21} & A_{22} \\
\end{array}
\times
\begin{array}{cc}
B_{11} & B_{12} \\
B_{21} & B_{22} \\
\end{array}
=
\begin{array}{cc}
C_{11} & C_{12} \\
C_{21} & C_{22} \\
\end{array}
\]

Figure 1: Block-Recursive Definition of Matrix-Matrix Multiply

An example schematic description of Matrix-Matrix Multiply (MMM) is presented in Figure 1. The values for \( C_{ij} \) are:

\[
\begin{align*}
C_{11} &= A_{11} \times B_{11} + A_{12} \times B_{21} \\
C_{12} &= A_{11} \times B_{12} + A_{12} \times B_{22} \\
C_{21} &= A_{21} \times B_{11} + A_{22} \times B_{21} \\
C_{22} &= A_{21} \times B_{12} + A_{22} \times B_{22}
\end{align*}
\] (1)
Hierarchically Tiled Arrays (HTAs): dimensions 1 and 0

The base case of this recursion is scalar multiplication. A scheme to address the scalar elements of a (hierarchically tiled) array by indexing scheme is legal. This description is called block-recursive because the large multiplication has been decomposed into eight smaller multiplications; the base case of this recursion is scalar multiplication.

To demonstrate the features of the DSL for block-recursive matrix algorithms, we consider a more complex example: block triangular solve ($\Delta_{\text{solve}}$), which is part of the Level 3 BLAS (Basic Liner Algebra Subroutines). The essence of the problem is to solve the linear system

$$ L \times X = B, $$

where $L$ is an $n \times n$ lower triangular matrix and $X$ and $B$ are $n \times s$ matrices. The block-recursive diagram is presented in Figure 2.

Using the system of equations (1) we can construct the following specialized versions for $\Delta_{\text{solve}}$:

$$
\begin{align*}
L_{11} \times X_{11} &= B_{11} \\
L_{11} \times X_{12} &= B_{12} \\
L_{21} \times X_{11} + L_{22} \times X_{21} &= B_{21} \\
L_{21} \times X_{12} + L_{22} \times X_{22} &= B_{22}
\end{align*}
$$

Using this new system of equations we can write a high-level block-recursive description of the triangular solve problem $X = \Delta_{\text{solve}}(L, B)$:

$$
\begin{align*}
X_{11} &= \Delta_{\text{solve}}(L_{11}, B_{11}) \\
X_{12} &= \Delta_{\text{solve}}(L_{11}, B_{12}) \\
X_{21} &= \Delta_{\text{solve}}(L_{22}, B_{21} - L_{21} \times X_{11}) \\
X_{22} &= \Delta_{\text{solve}}(L_{22}, B_{22} - L_{21} \times X_{12})
\end{align*}
$$

We use the fact that $L_{11}$ and $L_{22}$ are lower triangular matrices, so we implicitly restrict them to be square, and thus we have to slice $L$ by the same amount in both directions. The base case is when all the matrices $L$, $X$ and $B$ have dimensions $1 \times 1$, and then the result is simply $X_{11} = B_{11}/L_{11}$. The DSL code is presented in Figure 3. It makes use of Hierarchically Tiled Arrays (HTAs):

function $X$: $n \times s = \Delta_{\text{Solve}}(L: n \times n, B: n \times s)$;
$\forall$ $p n$: $0 \leq p n \leq n$
$\forall$ $p s$: $0 \leq p s \leq s$
$L' = \text{tile}(L, [pn], [pn])$
$X' = \text{tile}(X, [pn])$
$B' = \text{tile}(B, [pn], [sn])$
$X'[1,1] = \Delta_{\text{solve}}(L'[1,1], B'[1,1])$
$X'[1,2] = \Delta_{\text{solve}}(L'[1,1], B'[1,2])$
$X'[2,1] = \Delta_{\text{solve}}(L'[2,2], B'[2,1] - L'[2,1] \times X'[1,1])$
$X'[2,2] = \Delta_{\text{solve}}(L'[2,2], B'[2,2] - L'[2,1] \times X'[1,2])$

function $X$: $1 \times 1 = \Delta_{\text{Solve}}(L: 1 \times 1, B: 1 \times 1)$
$(X(1,1) = B(1,1)/L(1,1))$

Figure 3: $\Delta_{\text{solve}}$ expressed in a sample DSL for LA codes

This description uses the $\text{tile}$ construct and the $Z(x, y)$ indexing constructs from HTAs. In summary, $X' = \text{tile}(X, [r_0, r_1, \ldots, r_n], [c_0, c_1, \ldots, c_n])$ is “tiling” the matrix $X$, by dividing it horizontally after rows $r_i, 0 \leq i \leq n$ and vertically after columns $c_i, 0 \leq i \leq m$. Therefore the result $X'$ is a tiled array with $n \times m$ elements (or “tiles”). The indexing scheme $X'[i, j]$ is used to refer to the specific tiles. Furthermore one can always use the normal “flat” indexing scheme to address the scalar elements of a (hierarchically tiled) array by $X'(i, j)$.
3 Generate an Iterative Version

Once the programmer provides a block-recursive high-level description of the kernel, the natural next step is to covert that description to an iterative form. A naïve iterative $\Delta_{solve}$ implementation is presented in Figure 4.

```c
for (int si = 0; si < s; si++)
    for (int ri = 0; ri < n; ri++)
    {
        double t = 0;
        for (int ci = 0; ci < ri; ci++)
            t += X[ci][si] * L[ri][ci];
        X[ri][si] = (B[ri][si] - t) / L[ri][ri];
    }
```

Figure 4: Naïve $\Delta_{solve}$ implementation

Rather than converting block-recursive descriptions directly to loop nests operating on scalar values, our goal is to covert to generalized tiled iterative versions, like the one presented in Figure 5. The + and $\times$ there, are the corresponding matrix operations. $L'$, $X'$ and $B'$ are suitably tiled matrices, which in this case means that the tiles of $L'$ are $N_b \times N_b$ in size and the tiles for $X'$ and $B'$ are $N_b \times S_b$ in size. $N_b$ and $S_b$ are called optimization parameters.

```c
#pragma search N_b
#pragma search S_b
for (int si = 0; si < s'; si += S_b)
    for (int ri = 0; ri < n'; ri += N_b)
    {
        double[N_b][S_b] T = zeros(N_b, S_b);
        for (int ci = 0; ci < ri; ci += N_b)
            T += L'[ri,ci] \times X'[ci,si];
        X'[ri,si] = $\Delta_{solve}$(L'[ri,ri], B'[ri,si] - T);
    }
```

Figure 5: Tiled iterative $\Delta_{solve}$ implementation

The code in Figure 5 represents a single tiling level and complex operations like Matrix-Multiply and $\Delta_{solve}$ are present within. We can apply this transformation recursively and thus generate multi-level tiling, which from an optimization perspective could correspond to optimizing for different levels of the memory hierarchy, including registers and parallelism (SMP and SMT). The last recursive application should be used in the “simple” case, this producing something like the code on Figure 4. In this way, we can get multiple tiling levels with only scalar operations in the generated code.

4 Apply Restructuring Optimizations

In this step we apply parameterized forms of well known high-level as well as low-level source code restructuring transformations and define to corresponding optimization parameters they depend on. The transformations include but are not limited to loop tiling, unrolling, software pipelining, register allocation, inline expansions, instructions scheduling, translation to/from novel data structures, reordering high-level operations, etc.

5 Determine Optimization Parameters

In this step we use modelling and empirical optimization techniques to determine suitable values for all optimization parameters identified during the block-recursive to tiled iterative translation. In our $\Delta_{solve}$ examples these are $N_b$ and $S_b$, but there can be additional ones to control optimizations like unrolling, software pipelining and so on.

6 Generate Library Code

Finally we invoke the back-end C compiler to generate library code.
7 Summary of Goals

As part of this project you will:

1. Design/Implement Recursive to Iterative conversion
2. Design/Implement Code Restructuring passes for ISSFX.CCK
3. Apply this approach to the BLAS kernels and demonstrate that in this way you can achieve performance identical to that of tools like ATLAS, while providing a more general framework for doing so.

This is a four person project!

8 References

Some examples of projects which involve similar ideas to a certain extend are:

1. FLAME (http://www.cs.utexas.edu/users/flame/);
2. FFT/W (http://www.fftw.org/)

You should take a look at the examples and the papers available through the web-sites above.