COL 729: Lab 3 - Experimenting with the polyhedral model

Prashant Agrawal - 2018CSZ8011

April 25, 2019

Contents

1 Running times

Table 1 shows the running times of the `gemm.c` benchmark of the polybench benchmark suite with different compilers. All runs were performed on an Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz machine with 6 cores with hyperthreading with the following compilation flags: `-DPOLYBENCH_TIME -DEXTRALARGE_DATASET -DDATA_TYPE_IS_INT`.

Remark: The compilation flag `-DDATA_TYPE_IS_INT` did not seem to have any effect on the data type of the matrices as the benchmark always picked up `double` as the matrices' data type. We therefore stick to performing all experiments on matrices of doubles for easier comparisons.

Compiler	Runtime (s)
`gcc -O3`	375.39
`clang -O3` (without polly)	374.89
`icc -O3`	18.32
`clang -O3 -mllvm -polly`	11.84

Table 1: Runtimes of `gemm.c` on different compilers

2 Analysis

2.1 `gcc -O3`

Listing 1 shows pseudo code for the execution logic in case of `gcc -O3`. It is clear that in this case the `O3` logic is exactly similar to the source code, with no significant optimizations performed to improve the data locality or parallelism, therefore resulting in poor overall performance.

```
for (i=0; i<4000; i++) {
  for (j=0; j<4000; j++) {
    C[i][j] *= beta;
    for (k=0; k<4000; k++) {
      C[i][j] += alpha * A[i][k].B[k][j];
    }
  }
}
```

Listing 1: Pseudo code for `gcc -O3 gemm.c`

2.2 `clang -O3` (without polly)

Listing 2 shows pseudo code for the execution logic in case of `clang -O3` (without polly). In this case, the only attempt at loop optimization is unrolling the innermost loop twice to save some jump instructions, but that is not enough and the performance in this case is almost as bad as `gcc -O3`.

```
for (i=0; i<4000; i++) {
  for (j=0; j<4000; j++) {
    C[i][j] *= beta;
    for (k=0; k<4000; k=k+2) {
      C[i][j] += alpha * A[i][k].B[k][j];
      C[i][j] += alpha * A[i][k+1].B[k+1][j];
    }
  }
}
```

Listing 2: Pseudo code for `clang -O3 gemm.c` (without polly)

2.3 `icc -O3`

Listing 3 shows pseudo code for the case of `icc -O3`. ICC performs many consequential optimizations:

First, it performs loop fission to calculate `C[i][j] *= beta` in a separate loop nest. This results in better data locality for both the loop nests.

Second, it realizes that since all the matrices `A`, `B` and `C` are stored in row-major form, any ordering of the loops would result in poor cache utilization for one of the matrices and would result in many

cache misses in the innermost loop. Therefore it performs its accesses in a tiling fashion to allow better cache locality along all three loop variables `i`, `j` and `k`.

Third, the loop in `j` is ordered to be the innermost so that it can enjoy great cache localization due to row major formats of both `C[i][j]` and `B[k][j]` (column-wise access of `A[i][k]` is moved to the outer loop).

Fourth, since the innermost loop is independent in `j`, it is vectorized using Intel's SSE instruction set extension using width 2 vectors, thus doubling the performance. In Sec. 2.5, we will see how we can exploit the hardware's capability to run even wider vectors, thus resulting in further performance improvement.

Finally, the innermost loop is unrolled a few times to save some condition checks and jump instructions.

```
for (i=0; i<4000; i++) {
  for (j=0; j<4000; j=j+8) {
    <C[i][j],   C[i][j+1]> *= <beta, beta>;
    ...
    <C[i][j+6], C[i][j+7]> *= <beta, beta>;
  }
}

for (bi=0; bi<32; bi++) {
  for (bj=0; bj<32; bj++) {
    for (bk=0; bk<32; bk++) {
      for (tk=0; tk<128; tk++) {
        k = 128*bk + tk;
        for (ti=0; ti<128; ti++) {
          i = 128*bi + ti;
          a = alpha*A[i][k];
          for (tj=0; tj<128; tj++) {
            j = 128*bj + tj;
            <C[i][j],   C[i][j+1]> += a*<B[k][j], B[k][j+1]>;
            ...
            <C[i][j+6], C[i][j+7]> += a*<B[k][j+6], B[k][j+7]>;
          }
        }
      }
    }
  }
}
```

Listing 3: Pseudo code for `icc -O3 gemm.c`

2.4 `clang -O3 -mllvm -polly`

Listing 4 shows pseudo code for the case of `clang -O3 -mllvm -polly`. Like ICC, Polly performs many loop transformations including loop tiling, reordering, fission and vectorization to achieve the best performance.

Polly first performs an aliasing analysis on the innermost loop statement to determine if above loop transformations are even applicable. It employs a memory bounds check to see if the range of memory locations accessed by matrix `C` overlaps with that accessed by matrices `A` or `B`, and switches to an unoptimized, non-vectorized version if this check fails at runtime. The pseudo code for this unoptimized version is the same as that obtained by `clang -O3` without polly (i.e. Listing 2).

If the memory bounds check passes, the code generated by Polly is similar to the one generated by `icc` in that it also relies on tiling for better data locality and cache utilization. The interesting thing in Polly is how it determines the sizes of the tiles to allow maximum cache utilization.

```
for (bi=0; bi<125; bi++) {
  for (ti=0; ti<32; ti++) {
    i = 32*bi + ti;
    for (bj=0; bj<125; bj++) {
      j = 32*bj;
      <C[i,j], C[i,j+1]>     *= <beta, beta>;
      ...
      <C[i,j+30], C[i,j+31]> *= <beta, beta>;
    }
  }
}

for (bj=0; bj<4; bj++) {
  for (bk=0; bk<10; bk++) {
    for (bi=0; bi<63; bi++) {
      for (tj=0; tj<1000; tj=tj+2) {
        j =  1000*bj + tj;
        if (j>3997) break;
        for (ti=0; ti<64; ti=ti+4) {
          i = 64*bi + ti;
          if (i>3996) break;
          for (tk=0; tk<400; tk++) {
            k = 400*bk + tk;
            if (k>3999) break;
            <C[i,j],     C[i,j+1]> += A[i][k] * <B[k,j], B[k,j+1]>;
            ...
            <C[i+3,j], C[i+3,j+1]> += A[i+3,k] * <B[k,j], B[k,j+1]>;
          }
        }
      }
    }
  }
}
```

Listing 4: Pseudo code for `clang -O3 -mllvm -polly gemm.c`

In the sequel, we try to analyze the rough number of cache misses in the above program. For simplicity, we focus our attention to the cache misses in one block only, i.e. within the innermost 3 loops. Let us assume that the cache size is C bytes and the cache line size is c bytes. Let's denote the sizes of the inner loops by $|t_i|$, $|t_j|$ and $|t_k|$ and the vector width by w (in the above program,

$|t_i| = 64$, $|t_j| = 1000$, $|t_k| = 400$ and $w = 2$). Finally, recall that all the matrices are made up of `double`s and are stored in row-major format.

Consider matrix `A`. Its innermost loop access pattern is such that the number of cache misses in it is $4.\frac{|t_k|\times 8}{c}$. For the `ti` loop, the total number of cache misses is $|t_i|.\frac{|t_k|\times 8}{c}$. If all entries of `A` accessed within the `tj` loop can live within the cache, then the total number of cache misses in the `tj` loop is the same as above (otherwise it is a multiple of $|t_j|$). That is, if $|t_i| \times |t_k| \times 8 < C$, then the total number of cache misses due to `A` is $|t_i|.\frac{|t_k|\times 8}{c}$ per block.

Now consider matrix `B`. `B`'s innermost loop access pattern is such that the number of cache misses in it is $|t_k| + |t_k|.\frac{w}{c}$ ($\frac{w}{c}$ represents integer division). If all the entries of `B` accessed within the `ti` loop can live within the cache, then the number of cache misses in the `ti` loop is the same as above, i.e. if $|t_k| \times w \times 8 < C$ then the number of cache misses in the `ti` loop is $|t_k| + |t_k|.\frac{w}{c}$, and therefore the total number of cache misses in the `tj` loop (i.e. one block) is $\frac{|t_j|}{w}.(|t_k| + |t_k|.\frac{w}{c})$.

Consider matrix `C`. We can analyze this assuming that the memory operations for `C` are moved outside the innermost loop. The total number of cache misses in the `ti` loop are $|t_i|+|t_i|\frac{w}{c}$. The total number of cache misses per block, i.e. total number of cache misses in the `tj` loop are $\frac{|t_j|}{w}(|t_i| + |t_i|\frac{w}{c})$.

Notice therefore that the tile sizes of loops (i.e. $|t_i|$, $|t_j|$ and $|t_k|$) and vector width w are governed by various constraints that ensure that a) entries within a tile fit within the cache, and b) the total number of cache misses is minimum. Note, however, that the vector width w doesn't need to be upper bounded to 2 for better cache utilization. This is something which we exploit in our hand-optimized version (Sec. 2.5).

2.5 Hand-optimized matrix multiplication

A hand-optimized matrix multiplication code has been attached with this submission (filename `gemm-handopt.c`). It uses Intel Intrinsics to embed advanced assembly-level vectorization instructions in C code. It builds upon the polly generated loop tilings and uses the fact that the experimental machine supports AVX-512 extension, thus presenting opportunity for wider vectorization. We use the maximum possible vector width (i.e. 8) supported by AVX-512 and that gives us almost 4x speed-up compared to Polly as expected (see Table 2).

A few other tricks that were used to boost the performance are in order. We used the fused-multiply-add (FMA) instructions in the innermost loop as they are very well suited for our application. The store/load from matrix `C` were moved outside the innermost loop to prevent many memory operations in the innermost loop.

The performance of this hand-optimized version is shown in Table 2 and a pseudo code (of the second loop nest only) is shown in Listing 5.

```
for (bj=0; bj<4; bj++) {
  for (bk=0; bk<10; bk++) {
    for (bi=0; bi<63; bi++) {
      for (tj=0; tj<1000; tj=tj+8) {
        j = 1000*bj + tj;
        if (j>3992) break;
        for (ti=0; ti<64; ti=ti+4) {
          i = 64*bi + ti;
```

```
            if (i>3996) break;
            for (tk=0; tk<400; tk++) {
              k = 400*bk + tk;
              if (k>3999) break;
              <C[i,j]...    C[i,j+7]> += A[i,k]*<B[k,j]...B[k,j+7]>;
              ...
              <C[i+3,j]...C[i+3,j+7]> += A[i+3,k]*<B[k,j]...B[k,j+7]>;
            }
          }
        }
      }
    }
}
```

Listing 5: Pseudo code for hand-optimized version of matrix multiplication

Compiler/Source	Runtime (s)
`gcc -O3 gemm.c`	375.39
`clang -O3 gemm.c` (without polly)	374.89
`icc -O3 gemm.c`	18.32
`clang -O3 -mllvm -polly gemm.c`	11.84
`icc -O3 gemm-handopt.c` (hand-optimized)	4.60

Table 2: Runtimes of `gemm.c` on different compilers

3 Polly - Key Strengths

- **Powerful tiling/loop interchange passes:** We have seen above that Polly is able to reorder loops, perform loop fission and loop tiling transforms to exploit the data locality the most. Moreover, the tile sizes seem to be carefully calculated so as to optimally utilize the cache. It is also able to insert vectorized instructions wherever possible, although as noted above it doesn't utilize the full vector width supported by the hardware.
- **Aliasing support:** Consider the example shown in Listing 6. Pointer `ptr` is assigned to either `A` or `B` depending on some non-affine condition that cannot be checked at compile time but is a tautology nonetheless leading to `ptr = A` always at runtime. The loop nest is such that when `ptr = A`, the `i` loop could be trivially optimized away since there exists no dependency on `i`. However, when `ptr = B`, a dependency chain in `B` is formed, which must be preserved by retaining the loop as it is.

  In cases like this, Polly does the right thing even if it can't determine the exact value of `ptr` at compile time. It inserts a memory bounds check which allows it to use the optimized loop nest in case the memory locations accessed by `ptr` don't overlap with that of `B` at runtime, and use the vanilla loop nest in case there is some aliasing in the memory locations pointed by `ptr` and those pointed by `B`.

  ...

```
double* ptr = (k+k*k >= k) ? A : B;

for (i=0; i<10000; i++) {
  for (j=0; j<10000; j++) {
    B[j] = ptr[j] + 5;
  }
}
```

Listing 6: Aliasing in Polly

- **Miscellaneous:** In addition, there are many benefits from a software engineering point of view. Polly works on LLVM IR, thus making it useful for a variety of high-level languages, not just C/C++. Also, its modular design allows it to be used with external optimizers.

4 Polly - Limitations

- **Vectorization of loops:** As shown in Sec. 2.4, Polly doesn't always utilize the full vector width supported by the hardware. This could be exploited to further boost performance as shown in our hand-optimized version (Sec. 2.5).

  In addition, in some programs, a loop is not vectorized because a tautology is not apparent at compile time. Consider Listing 7 where the condition `i+j>=i` is not checkable at compile time, and thus, a loop nest which was trivially vectorizable isn't vectorized at all.

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    if (i+j>=i) {
        C[i] = 0;
    } else {
        C[i] = C[i-1] + 1;
    }
  }
}
```

Listing 7: Vectorization of loops in Polly

- **Basic blocks are indivisible:** When performing its loop transformation passes, Polly considers basic blocks as indivisible blocks and thus can lose out on some parallelization/code motion opportunities if various statements within a basic block are independent. Consider the example shown in Listing 8 where the accesses to array `D` create a dependency chain. Since Polly treats the entire basic block as one entity, it loses the opportunity to parallelize the 4 sequential accesses of `C` even though all of them are independent in themselves and with `D`.

```
for (i=0; i<N; i=i+4) {
  C[i]   = A[i] * B[i];
  C[i+1] = A[i+1] * B[i+1];
  C[i+2] = A[i+2] * B[i+2];
  C[i+3] = A[i+3] * B[i+3];
  D[i]   = D[i-1] + 1;
```

```
}
```

Listing 8: An unrolled loop cannot be vectorized

- **Memset/memcpy based array accesses:** Consider the example in Listing 9 where the loop body is a statement equivalent to the statement `C[j] = A[j] + B[j]` and yet, while the latter would be easily optimized by Polly to remove the outer `i` loop, the former is left as it is.

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    memset(C+j, A[j] + B[j], sizeof(int));
  }
}
```

Listing 9: Memset based array access