On the difference between parallel loops and vector loops

N3735 2013-09-02

Robert Geva

Context

- This is not a proposal, it is a clarification of existing proposals.
- Some feedback regarding the loops proposed within the Cilk Plus proposal expressed interest in a loop that is both parallel and vector
- Other proposals pursue the idea that an algorithm can be expressed in a way that is independent of whether it is sequential, parallel or vector, and those are chosen as "execution policy"
- While all the above are useful and encouraged, this presentation suggests that single threaded vector loops are more foundational and of higher priority than the combined loops and algorithms.
- The fundamental argument is that vector loops and parallel loops are mostly very different, whereas their similarities are mostly superficial

The obvious

- Writing a parallel program includes:

1. Identifying tasks that can execute in parallel
2. Mapping work to the parallelism in the HW

The obvious

- Writing a parallel program includes:

1. Identifying tasks that can execute in parallel
   - This is sometimes hard
   - Easier for embarrassingly parallel problems
   - Hopefully machine independent
   - Need to get rid of data races
   - Express intent for parallel execution
2. Mapping work to the parallelism in the HW

The obvious

- Writing a parallel program includes:

1. Identifying tasks that can execute in parallel
2. Mapping work to the parallelism in the HW
   - w/o parallel HW there can be no parallel programming
   - Sometime just expressing intent delivers good performance
   - There is an interest in ability to separate concerns between expressing parallelism and mapping to HW resources (different expertise)
   - Mapping to HW includes cache efficiency, topology, data layout, etc

The Obvious

- It is nice if a programmer can just “express intent” and get acceptable performance
- Arguably
  - The state of parallel programming is not there
  - Even if it were close enough, the language has to support those programmers that have to get it there
- Conclusion: Mapping a parallel algorithm to the parallel resources in the HW is a necessary part of parallel programming
  - And in most cases, what makes parallel programming hard

Language vs. Performance perspective

- Language perspective:
  - Both parallel loop and vector loop provide permissions to execute the loop in a non sequential order
  - The difference between the two is hair splitting
  - Writing a loop once and experimenting with different intents is appealing
  - A programmer who doesn't care about splitting hairs should be able write a single loop (sometimes with multiple indices) to express intent for both and allow the compiler to map execution to the HW
- Performance perspective:
  - The code that can be efficiently parallelized is not the same as the code that can be efficiently vectorized
  - HW mapping of parallel loops is different from HW mapping of vector loops
  - Writing a loop once and changing the intent invalidates the performance work
  - Different loops in a loop nest maps account for different HW considerations

Catego ry	Description	Parallel Loops	Vector Loops
Seman tics	Countable loops	Yes	Yes
	Allow critical sections	Yes	No
	Allow data dependencies	No	Yes
	Allow C++ EH	Yes	No
Capabi lities	Allow inner parallel / vector loops	Yes	No
	Allow non commutative reductions	Yes	No
	Ability for a worker to invoke work on a different worker	Yes	No
	Multiple scheduling policies possible	Yes	No
Perfor mance	Efficient control flow divergence	Yes	No
engine ering	Efficient memory access divergence	Yes	No
	No overhead data communication between active workers	No	Yes
	Consideration for multi level topologies (HT, cores, sockets)	Yes	No
	Consideration for work migrating to different execution units	Yes	No
	Cache efficiency	Yes	Yes

Divide and conquer parallelism: Parallelizing Matrix Multiplication

C+= A*B

```
for (int i=0; i<n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i*n+j] += A[i*n+k] * B[k*n+j];
        }
    }
}
```

Parallelizing Matrix Multiplication

C+=A*B

```
cilk_for (int i=0; i<n; i++) {
  cilk_for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
          C[i*n+j] += A[i*n+k] * B[k*n+j];
      }
  }
}
```

Parallel, but not cache efficient

Parallelize the loop implementation by parallelizing the loop

Matrix multiplication using divide and conquer

C11	C12
C21	C22

=

A11	A12
A21	A22

×

B11	B12
B21	B22

```
A11 A12   B11 B12     A11*B11+A12*B21 , A11*B12+A21*B22
         *          =
A21 A22   B21 B22     A21*B11+A22*B21 , A21*B12+A22*B22
```

The identity leads to a recursive implementation that subdivides the matrix into quadrants

Parallelizing using divide and conquer

```
void mm_recurs (double *c, double *A, double *B,
 int n, int length)
 { if (length < threashold) do_loop();
     else {
        int mid = length / 2;
        double *C11 = C                 ;
        double *C12 = C           + mid;
        double *C21 = C + n*mid         ;
        double *C22 = C + n*mid + mid;
        double *A11 = A                 ;
        double *A12 = A           + mid;
        double *A21 = A + n*mid         ;
        double *A22 = A + n*mid + mid;
        double *B11 = B                 ;
        double *B12 = B           + mid;
        double *B21 = B + n*mid         ;
        double *B22 = B + n*mid + mid;
    // cont …
```

The result of parallelizing a loop does not even have to be a loop

Parallelizing using divide and

```
// cont …
cilk_spawn mm_recurs(C11, A11,B11,n,mid);
cilk_spawn mm_recurs(C12, A11,B12,n,mid);
cilk_spawn mm_recurs(C21, A21,B11,n,mid);
           mm_recurs(C22, A21,B12,n,mid);
cilk_sync;
cilk_spawn mm_recurs(C11, A12,B21,n,mid);
cilk_spawn mm_recurs(C12, A12,B22,n,mid);
cilk_spawn mm_recurs(C21, A22,B21,n,mid);
           mm_recurs(C22, A22,B22,n,mid);

}
```

The base case is used when the dimensions are small. It uses single treaded vector loop. The vector loop looks similar to the original, sequential loop.

Matrix Multiplication: Tiled version

```
Cilk_for (int r = 0; r < height; r += TILE_m) {
    cilk_for (int c = 0; c < n; c += TILE_n) {
        FTYPE atile[TILE_m][TILE_k], btile[TILE_n], ctile[TILE_m][TILE_n];

        ctile[:][:] = 0.0;
        for (int t = 0; t < k; t += TILE_k) {

                atile[:][:] = A[r:TILE_m][t:TILE_k];

                for (int rc = 0; rc < TILE_k; rc++) {
                    btile[:] = B[t+rc][c:TILE_n];

                    for (int rt = 0; rt < TILE_m; rt++) {
                        ctile[rt][:] += atile[rt][rc] * btile[:];
                    }
                }
            }

            C[r:TILE_m][c:TILE_n] = ctile[:][:];
        }
    }
}
```

Stencil example – serial loop

```
void loop_stencil(int t0, int t1, int x0, int x1, int y0, int y1, int z0, int z1)
{
    for(int t = t0; t < t1; ++t) {
        for(int z = z0; z < z1; ++z) {
            for(int y = y0; y < y1; ++y) {
                float c0 = coef[0], c1 = coef[1], c2 = coef[2], c3 = coef[3], c4 = coef[4];
                int s = z * Nxy + y * Nx;
                float * A_cur = &A[t & 1][s];
                float * A_next = &A[(t + 1) & 1][s];
                for(int x = x0; x < x1; ++x) {
                    float div = c0 * A_cur[x]
                        + c1 * ((A_cur[x + 1] + A_cur[x - 1])
                                + (A_cur[x + Nx] + A_cur[x - Nx])
                                + (A_cur[x + Nxy] + A_cur[x - Nxy]))
                        + c2 * ((A_cur[x + 2] + A_cur[x - 2])
                                + (A_cur[x + sx2] + A_cur[x - sx2])
                                + (A_cur[x + sxy2] + A_cur[x - sxy2]))
                        + c3 * ((A_cur[x + 3] + A_cur[x - 3])
                                + (A_cur[x + sx3] + A_cur[x - sx3])
                                + (A_cur[x + sxy3] + A_cur[x - sxy3]))
                        + c4 * ((A_cur[x + 4] + A_cur[x - 4])
                                + (A_cur[x + sx4] + A_cur[x - sx4])
                                + (A_cur[x + sxy4] + A_cur[x - sxy4]));
                    A_next[x] = 2 * A_cur[x] - A_next[x] + vsq[s+x] * div;
                }
            }
        }
    }
}
```

A simple parallelization

```
void loop_stencil(int t0, int t1, int x0, int x1, int y0, int y1, int z0, int z1)
{
    for(int t = t0; t < t1; ++t) {
#pragma omp parallel for
        for(int z = z0; z < z1; ++z) {
            for(int y = y0; y < y1; ++y) {
                float c0 = coef[0], c1 = coef[1], c2 = coef[2], c3 = coef[3], c4 = coef[4];
                int s = z * Nxy + y * Nx;
                float * A_cur = &A[t & 1][s];
                float * A_next = &A[(t + 1) & 1][s];
#pragma simd
                for(int x = x0; x < x1; ++x) {
                    float div = c0 * A_cur[x]
                        + c1 * ((A_cur[x + 1] + A_cur[x - 1])
                                + (A_cur[x + Nx] + A_cur[x - Nx])
                                + (A_cur[x + Nxy] + A_cur[x - Nxy]))
                        + c2 * ((A_cur[x + 2] + A_cur[x - 2])
                                + (A_cur[x + sx2] + A_cur[x - sx2])
                                + (A_cur[x + sxy2] + A_cur[x - sxy2]))
                        + c3 * ((A_cur[x + 3] + A_cur[x - 3])
                                + (A_cur[x + sx3] + A_cur[x - sx3])
                                + (A_cur[x + sxy3] + A_cur[x - sxy3]))
                        + c4 * ((A_cur[x + 4] + A_cur[x - 4])
                                + (A_cur[x + sx4] + A_cur[x - sx4])
                                + (A_cur[x + sxy4] + A_cur[x - sxy4]));
                    A_next[x] = 2 * A_cur[x] - A_next[x] + vsq[s+x] * div;
                }
            }
        }
    }
}
```

No restructuring,
just expressing intent

A cache efficient loop is more involved

```
void loop_stencil(int t0, int t1 int x0, int x1, int y0, int y1, int z0, int z1)
{
#if __MIC__
        const int yb = 4;               // Blocking factor for y axis
        const int zb = 4;               // Blocking factor for z axis
#else
        const int yb = 16;              // Blocking factor for y axis
        const int zb = 16;              // Blocking factor for z axis
#endif
for(int t = t0; t < t1; ++t) {
#pragma omp parallel for collapse(2)
        for(int zo = z0; zo < z1; zo+=zb) {
            for(int yo = y0; yo < y1; yo+=yb) {
                int yu = std::min(yo+yb,y1);
                int zu = std::min(zo+zb,z1);
                for(int z=zo; z<zu; ++z) {
                    for(int y=yo; y<yu; ++y) {
                        float c0 = coef[0], c1 = coef[1], c2 = coef[2], c3 = coef[3], c4 = coef[4];
                        int s = z * Nxy + y * Nx;
                        float * A_cur = &A[t & 1][s];
                        float * A_next = &A[(t + 1) & 1][s];
                        #pragma simd
                        for(int x = x0; x < x1; ++x) {
                            float div = c0 * A_cur[x]
                                + c1 * ((A_cur[x + 1] + A_cur[x - 1])
                                        + (A_cur[x + Nx] + A_cur[x - Nx])
                                        + (A_cur[x + Nxy] + A_cur[x - Nxy]))
                                + c2 * ((A_cur[x + 2] + A_cur[x - 2])
                                        + (A_cur[x + sx2] + A_cur[x - sx2])
                                        + (A_cur[x + sxy2] + A_cur[x - sxy2]))
                                + c3 * ((A_cur[x + 3] + A_cur[x - 3])
                                        + (A_cur[x + sx3] + A_cur[x - sx3])
                                        + (A_cur[x + sxy3] + A_cur[x - sxy3]))
                                + c4 * ((A_cur[x + 4] + A_cur[x - 4])
                                        + (A_cur[x + sx4] + A_cur[x - sx4])
                                        + (A_cur[x + sxy4] + A_cur[x - sxy4]));
                            A_next[x] = 2 * A_cur[x] - A_next[x] + vsq[s+x] * div;
                        }
                    }
                }
```

Also considering topology

```
void loop_stencil(int t0, int t1, int x0, int x1, int y0, int y1, int z0, int z1)
{
#if __MIC__||__MIC2__
        const int yb = 8;               // Blocking factor for y axis
        const int zb = 8;               // Blocking factor for z axis
#else
        const int yb = 16;              // Blocking factor for y axis
        const int zb = 16;              // Blocking factor for z axis
#endif
    for(int t = t0; t < t1; ++t) {
#pragma omp parallel for collapse(2)
        for(int zo = z0; zo < z1; zo+=zb) {
            for(int yo = y0; yo < y1; yo+=yb) {
                crew::forall( zo, std::min(zo+zb,z1), [=](int z) {
                    int yu = std::min(yo+yb,y1);
                    for(int y=yo; y<yu; ++y) {
                        float c0 = coef[0], c1 = coef[1], c2 = coef[2], c3 = coef[3], c4 = coef[4];
                        int s = z * Nxy + y * Nx;
                        float * A_cur = &A[t & 1][s];
                        float * A_next = &A[(t + 1) & 1][s];
#pragma simd

                        for(int x = x0; x < x1; ++x) {
                            float div = c0 * A_cur[x]
                                + c1 * ((A_cur[x + 1] + A_cur[x - 1])
                                            + (A_cur[x + Nx] + A_cur[x - Nx])
                                            + (A_cur[x + Nxy] + A_cur[x - Nxy]))
                                + c2 * ((A_cur[x + 2] + A_cur[x - 2])
                                            + (A_cur[x + sx2] + A_cur[x - sx2])
                                            + (A_cur[x + sxy2] + A_cur[x - sxy2]))
                                + c3 * ((A_cur[x + 3] + A_cur[x - 3])
                                            + (A_cur[x + sx3] + A_cur[x - sx3])
                                            + (A_cur[x + sxy3] + A_cur[x - sxy3]))
                                + c4 * ((A_cur[x + 4] + A_cur[x - 4])
                                            + (A_cur[x + sx4] + A_cur[x - sx4])
                                            + (A_cur[x + sxy4] + A_cur[x - sxy4]));
                            A_next[x] = 2 * A_cur[x] - A_next[x] + vsq[s+x] * div;
                        }
                    }
                });
            }
```

Here, topology accounts for SMT.

The addition of topology resulted in change to the cache blocking.

Conclusion

- Parallel loops and vector loops serve different purposes
- The semantic differences are secondary
- The significant differences are the capabilities and performance profiles
  - Leading to different algorithmic choices
- Most programmers should be able to start with expressing intent
  - And tune only as needed

- However, a loop that both parallelizes and vectorizes may disallow the necessary differences in tuning the parallel loop differently from the vector loop

One more thing

- A very likely implementation of a parallel and vector loop requires the compiler to split into a hierarchy, using a “heuristically chosen constant”

- The programmer can get the same effect, but using a “carefully tuned constant”

- Additional downside of relying on the compiler include loss of information such as alignment, constant loop bounds etc

- The combined loops is likely a premature optimization, experience with actual practice may be required