Project Related Discussion

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

September 2014 PMPH Lecture Notes

1. Code Structure
2. CPU parallelization
3. Code Transformations
   - Reasoning About Parallelism: Privatization & Array Expansion
   - Creating CUDA Kernels via Loop Distribution
   - Various Optimizations, e.g., Coalesced Memory
4. Project Code: A Bit More Complex Due to a Seq Loop

Datasets

- Small: OUTER=16, NUM_X=32, NUM_Y=256, NUM_T=90
- Medium: OUTER=32, NUM_X=47, NUM_Y=181, NUM_T=93
- Large: OUTER=128, NUM_X=256, NUM_Y=256, NUM_T=128

The goal is to parallelize all three parallel dimensions, i.e., TRIDAG can be re-written based on segmented `scans`. Also optimize such that all accesses to global memory are coalesced.

Target primarily the *small* dataset, which, otherwise, does not has enough parallelism to fully utilize the hardware.

Code Structure

Code Entry Point

```
void   run_OrigCPU(...) {
  REAL strike;
  PrivGlobs globs(numX,numY,numT);
  for(int i=0; i<outer; ++ i) {
    strike = 0.001*i;
    res[i] = value( globs,s0,strike,t,
                    alpha,nu,   beta,
                    numX, numY, numT );
  }
}
```

```
REAL   value( ... ) {
  initGrid(s0,alpha,nu,t,
           numX,numY,numT,
           globs);
  initOperator(globs.myX,
               globs.myDxx);
  initOperator(globs.myY,
               globs.myDyy);
  setPayoff(strike, globs);
  for(int i=numT-2;i>=0;--i){
    updateParams(i,alpha,beta,
                 nu,globs);
    rollback(i, globs);
  }
  return globs.myResult[globs.myXindex]
                       [globs.myYindex];
}
```

Loop Nests

Loop Nests

```
rollback( ... ) {
  vector<vector<REAL> > u(numY, vector<REAL>(numX));   // [numY][numX]
  vector<vector<REAL> > v(numX, vector<REAL>(numY));   // [numX][numY]
  vector<REAL> a(numZ), b(numZ), c(numZ), y(numZ);     // [max(numX,numY)]
  vector<REAL> yy(numZ);  // temporary used in tridag  // [max(numX,numY)]
  for(i=0;i<numX;i++) {
    for(j=0;j<numY;j++) {
      u[j][i] = dtInv*globs.myResult[i][j];
  } }  .......
  // implicit y
  for(i=0;i<numX;i++) {
    for(j=0;j<numY;j++) {  // here a, b, c should have size [numY]
      a[j] =  - 0.5*(0.5*globs.myVarY[i][j]*globs.myDyy[j][0]);
      b[j] = dtInv - 0.5*(0.5*globs.myVarY[i][j]*globs.myDyy[j][1]);
      c[j] =  - 0.5*(0.5*globs.myVarY[i][j]*globs.myDyy[j][2]);
    }
    for(j=0;j<numY;j++)
      y[j] = dtInv*u[j][i] - 0.5*v[i][j];
    // here yy should have size [numY]
    tridag(a,b,c,y,numY,globs.myResult[i],yy);
} }
```

How To Parallelize

- summarize accesses inter-procedurally. For each loop what does it write and what does it read?
- Within each loop: are all reads covered by writes executed within the same iteration? If so then privatization solves those dependencies!
- For CUDA: do array expansion instead of privatization.
- Decide for each loop whether it can or cannot be parallelize.
- Use loop distribution to create perfect nests, which will become later your CUDA kernels.
- Use loop interchange and/or matrix transposition to obtain coalesced access to global memory.

CPU Parallelization

In function `run_OrigCPU` move the declaration of `strike` and `globs` inside the loop, and parallelize the loop via an OpenMP pragma:

Parallelizing the Outermost Loop Via OpenMP

```
#pragma omp parallel for default(shared) schedule(static)
   for( unsigned i = 0; i < outer; ++ i ) {
       REAL strike;
       PrivGlobs    globs(numX, numY, numT);

       strike = 0.001*i;
       res[i] = value( globs, s0, strike, t,
                       alpha, nu,    beta,
                       numX,  numY,  numT );
   }
```

Explain why this is safe in the report!
(For example if you do NOT move the declarations inside the loop and still parallelize the loop, the execution will NOT validate).

Reasoning About Parallelism: Privatization

Parallelizing the Outermost Loop Via Privatization

```
float A[N];
for(int i=0;i<M;i++){ //seq
  for(int j=0;j<N;j++){
    A[j] = ...
  }
  ...
  for(int j=0;j<N;j++){
    ... = A[j]
  }
}
```

```
for(int i=0;i<M;i++){ //par
  float A[N];
  for(int j=0;j<N;j++){
    A[j] = ...
  }
  ...
  for(int j=0;j<N;j++){
    ... = A[j]
  }
}
```

- The outermost loop of index `i` is NOT parallel as it is, because all its iterations write and read all indices of array `A`.
- However, the iteration reads (is covered by) what was written in the same iteration, a.k.a., array `A` can be privatized.
- Privatization can be achieved by moving the declaration of a inside the outermost loop (each iteration works with its own private version of `A`).

Array Expansion

Semantically Equivalent: Privatized *vs* Expanded A

```
for(int i=0;i<M;i++){ //par
  float A[N];
  for(int j=0;j<N;j++){
    A[j] = ...
  }
  ...
  for(int j=0;j<N;j++){
    ... = A[j]
  }
}
```

```
float A[M, N];
for(int i=0;i<M;i++){ //par
  for(int j=0;j<N;j++){
    A[i,j] = ...
  }
  ...
  for(int j=0;j<N;j++){
    ... = A[i,j]
  }
}
```

- In CUDA it is preferable that all memory is allocated before the kernel starts, hence making array `A` local would not work.
- Instead, expand array `A` with an extra (outermost) dimension, whose size is the count of the outermost loop.
- Now iteration `i` has exclusive access, i.e., writes to and reads from, row `i` of expanded array `A`.
- The two versions of code below are semantically equivalent!

Create CUDA Kernels via Loop Distribution

- Theorem: A parallel loop can be distributed across its statements (guaranteed that its dependency graph does not have cycles).
- CUDA kernels are obtained by distributing the outer loop around the inner loops (in order to improve the degree of parallelism).

Degree of parallelism M *vs.* **Degree of parallelism: M*N**

```
float A[M, N];
for(int i=0;i<M;i++){ //par
  for(int j=0;j<N;j++){
    A[i,j] = ...
  }
  ...
  for(int j=0;j<N;j++){
    ... = A[i,j]
  }
}
```

```
float A[M, N];
for(int i=0;i<M;i++){ //par
  for(int j=0;j<N;j++){ //par
    A[i,j] = ...; // 2D CUDA kernel
} }
for(int i=0;i<M;i++){ //par
  ...
}
for(int i=0;i<M;i++){ //par
  for(int j=0;j<N;j++){ //par
    ... = A[i,j]; // 2D CUDA Kernel
} }
```

Inline Simple Expression vs Array Expansion

- Loop distribution requires array expansion of the local variables.
- If the local variable is a simple scalar expression it is better to inline that expression rather than creating an array for it.
- Use your better judgment when to distribute and when to inline, i.e., do not create too many arrays (tradeoff between redundant computation AND extra memory & global accesses)

Inline Scalar Variables Rather Than Array Expansion

```
float A[M, N];
for(int i=0;i<M;i++){ //par
  int tmp = i*i;
  for(int j=0;j<N;j++){
    A[i,j] = ... * tmp;
  }
} ↓   inline scalar exp  ↓
float A[M, N];
for(int i=0;i<M;i++) //par
  for(int j=0;j<N;j++){//par
    A[i,j] = ... * ((float)i*i);
  }
```

```
// Systematic distribution will create
// Many Arrays, and Many access to Global
// Memory. (It might be cheaper to do some
// ← redundant computation instead).
float tmps[M];
float A[M, N];
for(int i=0;i<M;i++) //par
  tmps[i] = (float)(i*i);
for(int i=0;i<M;i++) //par
  for(int j=0;j<N;j++){//par
    A[i,j] = ... * tmps[i];
  }
```

1. Code Structure
2. CPU parallelization
3. Code Transformations
   - Reasoning About Parallelism: Privatization & Array Expansion
   - Creating CUDA Kernels via Loop Distribution
   - Various Optimizations, e.g., Coalesced Memory
4. Project Code: A Bit More Complex Due to a Seq Loop

Optimising CUDA Kernel (Memory Coalescing)

- After creating the CUDA Kernels, one might also want to optimise them, for example
- Coalesced Access to global memory may be obtained via loop interchange or (segmented) matrix transposition.

Coalesced Access via Loop Interchange or Matrix Transposition

```
float A[M, N];
for(int j=0;j<N;j++){ //par
  for(int i=0;i<M;i++){ //par
    A[i,j] = ... // uncoalesced access
} } ↓   loop interchange   ↓

for(int i=0;i<M;i++){ //par
  for(int j=0;j<N;j++){ //par
    A[i,j] = ... // coalesced access
} }
```

```
// Fixing uncoalesced accesses via
// matrix transposition
float Atr[N, M];
for(int j=0;j<N;j++){ //par
  for(int i=0;i<M;i++){ //par
    Atr[j,i] = ... // coalesced
} }
float A[M,N];
A=transpose(Atr); //Atr[j,i]≡A[i,j]
```

Note that applying loop interchange may make some uncoalesced accesses coalesced, but it might also make other (originally) coalesced accesses uncoalesced. In those cases use TRANSPOSITION.

You May Need Segmented Transpose

- Project uses three dimensional arrays, i.e., an array of matrices, and requires transposing each matrix (the two inermost dims).
- Nothing to be afraid of – this corresponds to a three dimensional CUDA kernel in which you write the matrix-transposition code for the innermost two dimensions. Pseudocode below:

Segmented Transposition: Sequential and CUDA Kernel

```
float A[O, M, N];
for(int k=0;k<O;k++){ //par
  for(int i=0;i<M;i++){ //par
    for(int j=0;j<N;j++){ //par
      Atr[k,j,i] = A[k,i,j];
} } }
```

```
__global__ void sgmMatTranspose( float* A,
      float* trA, int rowsA, int colsA ) {
  __shared__ float tile[T][T+1];
  int gidz=blockIdx.z*blockDim.z*threadIdx.z;
  A+=gidz*rowsA*colsA; Atr+=gidz*rowsA*colsA;
  // follows code for matrix transp in x & y
  int tidx = threadIdx.x, tidy = threadIdx.y;
  int j=blockIdx.x*T+tidx,i=blockIdx.y*T+tidy;
  if( j < colsA && i < rowsA )
    tile[tidy][tidx] = A[i*colsA+j];
  __syncthreads();
  i=blockIdx.y*T+tidx; j=blockIdx.x*T+tidy;
  if( j < colsA && i < rowsA )
    trA[j*rowsA+i] = tile[tidx][tidy];
}
```

Sequential Loop in Between Parallel Loops

- Loop of index t is sequential because it reads the array myResult[k,0:M-1,0:N-1] produced by previous iteration t-1!
- Distribute the outermost loop across the two loop nests, then interchange to get loop of index t in the outermost position:

Code after Array Expansion

```
float myResult[Outer, M, N];
for(int k=0;j<Outer;k++){ //par
  for(int i=0; i<M; i++) //par
    for(int j=0; j<N; j++) //par
      myResult[k,i,j] = ...;

  for(int t=0;t<T;t++){ //seq
    for(int i=0; i<M; i++) //par
      for(int j=0; j<N; j++) //par
        ... = ... myResult[k,i,j] ... ;
    for(int i=0; i<M; i++) //par
      for(int j=0; j<N; j++) //par
        myResult[k,i,j] = ...;
  }
}
```

Get Seq Loop Outside

```
float myResult[Outer, M, N];
for(int k=0;j<Outer;k++){ //par
  for(int i=0; i<M; i++) //par
    for(int j=0; j<N; j++) //par
      myResult[k,i,j] = ...;
}
for(int t=0;t<T;t++){ //seq
  for(int k=0;j<Outer;k++){ //par
    for(int i=0; i<M; i++) //par
      for(int j=0; j<N; j++) //par
        ... = ... myResult[k,i,j] ...;
    for(int i=0; i<M; i++) //par
      for(int j=0; j<N; j++) //par
        myResult[k,i,j] = ...;
} }
```

Sequential Loop in Between Parallel Loops

- Finally, distribute again loop `k` against the two inner loop nests to create CUDA kernels! Then optimize coalescing, etc.!

After Distrib Kernels 2 & 3 are called inside a Sequential Loop

```
float myResult[Outer, M, N];
for(int k=0;j<Outer;k++){ //par
  for(int i=0; i<M; i++) //par
    for(int j=0; j<N; j++) //par
      myResult[k,i,j] = ...;
}
for(int t=0;t<T;t++){ //seq
  for(int k=0;j<Outer;k++){ //par
    for(int i=0; i<M; i++) //par
      for(int j=0; j<N; j++) //par
        ... = ... myResult[k,i,j] ... ;
    for(int i=0; i<M; i++) //par
      for(int j=0; j<N; j++) //par
        myResult[k,i,j] = ...;
} }
```

```
float myResult[Outer, M, N];
for(int k=0;j<Outer;k++){  //Kernel1
  for(int i=0; i<M; i++)   //Kernel1
    for(int j=0; j<N; j++) //Kernel1
      myResult[k,i,j] = ...;
}
for(int t=0;t<T;t++){ //seq
  for(int k=0;j<Outer;k++){ //Kernel2
    for(int i=0; i<M; i++)  //Kernel2
      for(int j=0; j<N; j++)//Kernel2
        ... = ... myResult[k,i,j] ... ;
  }
  for(int k=0;j<Outer;k++){ //Kernel3
    for(int i=0; i<M; i++)  //Kernel3
      for(int j=0; j<N; j++)//Kernel3
        myResult[k,i,j] = ...;
} }
```