CS639 Parallel and Throughput-Optimized Programming
Spring Semester 2020

Practice Midterm Exam

March 3, 2020

Time: 2 hrs

1. [40% = 8 questions × 5% each] MULTIPLE CHOICE SECTION. Circle or underline the correct answer (or answers). You do not need to provide a justification for your answer(s).

(1) One of the possible implementations of a 2D Laplacian stencil we saw in class was as follows:

```
void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM]) {
#pragma omp parallel for
  for (int j = 1; j < YDIM-1; j++) // iterate on Y-dimension
  for (int i = 1; i < XDIM-1; i++) // then iterate on X-dimension
    Lu[i][j] = -4 * u[i][j] + u[i+1][j]
      + u[i-1][j] + u[i][j+1] + u[i][j-1];
}
```

This particular variant did not perform as well as others; what was the reason for this suboptimal performance? **(Circle or underline the ONE most correct answer)**

(a) There is not enough parallelism to exploit using OpenMP and multithreading.
(b) The 2D arrays involved are being traversed in the opposite order compared to how they are stored in memory.
(c) This is a reduction operation, and we have not declared it as such to OpenMP.

(2) Which of the following would be examples of *reduction* operations?
(Circle or underline ALL correct answers)

(a) Computing the maximum value among all entries of an array.
(b) Subtracting a constant number from every entry of an array.
(c) Computing the sum of all elements of an array.

(3) In class, we saw 2 methods of applying a Laplacian stencil on a 2D/3D array. The first method (a variant of which appears in question 1, above), does not use a matrix, but instead expresses the stencil application in code (we will call this the *matrix-free* approach). The second method, seen later in class, constructed the Laplace matrix explicitly, and applied it as a matrix-vector multiplication (we will call this the *matrix-based* method). Which of the following statements are true?
(Circle or underline ALL correct answers)

(a) The matrix-based approach is not able to take advantage of caching, since every value involved in a matrix-vector multiplication is read or written just once.
(b) The matrix-free approach typically has greater potential for performance, since it doesn't need to pay the memory access cost of reading a matrix from memory, and this can be a significant part of the runtime cost.
(c) Even if the two approaches might have different runtime cost, they can both take advantage of parallelism to a certain extent.

(4) Assume we have a lower-triangular matrix $\mathbf{L}$ stored in Compressed Sparse Row format. Which of the following operations would be the easiest to parallelize?
(Circle or underline the ONE most correct answer)

(a) Performing a forward-substitution on the system $\mathbf{Lx} = \mathbf{b}$, for a given right-hand-side vector $\mathbf{b}$.
(b) Computing the matrix-vector multiplication of $\mathbf{L}$ with a vector $\mathbf{x}$.
(c) Computing the matrix-vector multiplication of $\mathbf{L}^T$ (the transpose of this matrix) with a vector $\mathbf{x}$.

(5) Which factor among (a) the speed of performing arithmetic operations (i.e. "compute bandwidth") or (b) the speed of reading/writing data from/to memory (i.e. "memory bandwidth") would you say has the most impact on our ability to achieve good parallel performance on matrix-vector multiplications?
Note: You may assume that we are dealing with large, explicitly constructed matrices (as opposed to stencils applied via code). Most modern CPUs and GPUs can execute tens to hundreds of arithmetic operations in the time required for one memory access (from main memory).
(Circle or underline the ONE most correct answer)

(a) We are primarily constrained by compute bandwidth, since caching should help alleviate the cost of memory access.
(b) We are primarily constrained by memory bandwidth, since streaming the matrix from memory would be the main cost.
(c) For sparse matrices with fewer than 10-100 entires per row, we would be constrained by memory speed. For matrices with more entries per row than that, the burden will shift to compute speed.

(6) The Compressed Sparse Row format for matrices stores a sparse matrix in three arrays, which we have called `rowOffsets`, `columnIndices`, and `values`. Assume that the matrix we are storing is a square, $N \times N$ matrix. Which of the following statements are true?
(Circle or underline ALL correct answers)

(a) The length of the array `rowOffsets` is equal to N.
(b) The last entry of the array `rowOffsets` is equal to the total number of non-zero entries in the sparse matrix.
(c) If `rowOffsets[i] <= k < rowOffsets[i+1]`, and also `columnIndices[k] == j`, then `values[k]` will contain the value of the matrix at row `i` and column `j`.

(7) We are storing a lower-triangular matrix $\mathbf{L}$ in CSR format. Assume we know that all diagonal elements of this matrix are present in the sparsity pattern, i.e. none of them are zero. Where would we find the value of the diagonal element $\mathbf{L}_{ii}$?
(Circle or underline the ONE most correct answer)

(a) The value of $\mathbf{L}_{ii}$ is `values[i]`
(b) The value of $\mathbf{L}_{ii}$ is `values[rowOffsets[i]]`
(c) The value of $\mathbf{L}_{ii}$ is `values[rowOffsets[i+1]-1]`

(8) Consider the following snippet of code

```
float a[8], b[8], c[8];
for (int i = 0; i < 8; i++) {
  if ((i%2) == 0) c[i] = a[i] + b[i];
  if ((i%2) == 1) c[i] = a[i] * b[i];
}
```

We would like to consider using SIMD/Vector instructions (such as SSE, AVX, AVX2 or AVX-512) to parallelize this operation. However, in this case, this will not be so simple. Why?
(Circle or underline the ONE most correct answer)

(a) SIMD/Vector capabilities can *add* several numbers at once, but not *multiply.*
(b) Doing eight operations at once is beyond the vector capabilities of modern CPUs.
(c) SIMD/Vector computation is predicated on performing the *same* operation across all elements of arrays. An operation mix where some operations are additions, and some are multiplications is not a natural fit for vectorization (without modifications, at least).

2. [48% = 4 questions × 12% each] SHORT ANSWER PROBLEMS. Briefly answer each of the following questions. **It should take no more than just a few (2-3) sentences**.

   (a) Multiplying the *transpose* of a CSR matrix with a vector, as in the following code

```
void MatTransposeVecMultiply(CSRMatrix& mat, const float *x, float *y) {
   int N = mat.mSize;
   const auto rowOffsets = mat.GetRowOffsets();
   const auto columnIndices = mat.GetColumnIndices();
   const auto values = mat.GetValues();

   for (int i = 0; i < N; i++) y[i] = 0.;

   for (int i = 0; i < N; i++)
     for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
     const int j = columnIndices[k];
     y[j] += values[k] * x[i];
   }
}
```

   is not an easy kernel to parallelize (say, using OpenMP, without any further modifications). Can you explain what makes such parallelization difficult (say, as opposed to a normal matrix-vector multiply, without transposing the matrix)?

   (b) There is something wrong, or at least sub-optimal, with the following code (taken from our Laplace Solver examples)

```
float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM]) {
   double result = 0.;
#pragma omp parallel for
   for (int i = 1; i < XDIM-1; i++)
   for (int j = 1; j < YDIM-1; j++)
   for (int k = 1; k < ZDIM-1; k++)
     result += (double) x[i][j][k] * (double) y[i][j][k];
   return (float) result;
}
```

   Can you propose a fix to this? Can you explain what the consequences might have been if we left the code as given above?

(c) We are storing a sparse 5×5 matrix in CSR format, using the following representation:

```
rowOffsets[] = { 0, 3, 5, 8, 10, 12 };
columnIndices[] = { 0, 1, 3, 1, 2, 0, 1, 4, 1, 3, 0, 4 };
values[] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0 };
```

Can you write out this sparse matrix in the common mathematical notation?

(d) Consider the standard implementation of the Laplacian Stencil application as we saw in class:

```
void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM]) {
#pragma omp parallel for
  for (int i = 1; i < XDIM-1; i++)
            // <-- what if the #pragma omp parallel for was moved here?
  for (int j = 1; j < YDIM-1; j++)
    Lu[i][j] = -4 * u[i][j] + u[i+1][j]
      + u[i-1][j] + u[i][j+1] + u[i][j-1];
}
```

What do you believe might happen if we were to move the `#pragma omp parallel for` directive at the location that the comment indicates (i.e. in between the two for loops)? Would you generally expect improved/deteriorated/unchanged parallel performance? Explain.

3. [22% – Including 10% BONUS] Our implementation of a matrix-vector multiply using a CSR matrix was as follows:

```
void MatVecMultiply(CSRMatrix& mat, const float *x, float *y) {
   int N = mat.mSize;
   const auto rowOffsets = mat.GetRowOffsets();
   const auto columnIndices = mat.GetColumnIndices();
   const auto values = mat.GetValues();

#pragma omp parallel for
   for (int i = 0; i < N; i++)
     for (int k = rowOffsets[i]; k < rowOffsets[i+1]; k++) {
     const int j = columnIndices[k];
     y[i] += values[k] * x[j];
   }
}
```

Now, imagine that we had a Compressed Sparse **Column** (CSC) matrix, which we represented using a structure:

```
struct CSCMatrix {
   int mSize;
   std::unique_ptr<int> mColumnOffsets;
   std::unique_ptr<int> mRowIndices;
   std::unique_ptr<float> mValues;
   int* GetColumnOffsets() { return mColumnOffsets.get(); }
   int* GetRowIndices() { return mRowIndices.get(); }
   float* GetValues() { return mValues.get(); }
};
```

With this information implement either (just one of these – your choice) : (a) A matrix-vector multiplication that uses a CSC matrix, or (b) a matrix-vector multiplication that multiplies with the *transpose* of a CSC matrix.