Third and Fourth Weekly Assignment for the PMPH Course (21 pts total)

Hans Peter, pvr448

This is the text of the third and fourth weekly assignment for the PMPH edition 2024-2025.

- Assignment 3 consists of tasks 1, 2 and 4: total 10 pts (tasks 1 and 2 are pen and paper, task 4 is a programming task)
- Assignment 4 consists of tasks 3, 5 and 6: total 11pts (all are programming tasks)
- The due date is the same for both assignments; you are encouraged to submit once.
- The programming tasks can be solved in **groups of two** but the report writing and submission should be individual. Please write on the first page the names and ku-ids of all group members.

Hand in your solutions in the form of a short report in text or PDF format, along with the whole codebase that has been improved with your solutions. We hand-in incomplete code in the archive `w3-code-handin.tar.gz`. You are supposed to fill in the missing code such that all the tests are valid and to report performance results. Please implement the missing parts directly in the provided files and submit the whole code (**not** only the missing parts). There are comments in the source files that are supposed to guide you (together with the text of this assignment).

Unziping the handed in archive `w3-code-handin.tar.gz` will create the `w3-code-handin` folder, which contains:

- A `helper.h` file that contains helper functionality used by all exercises
- A `README.md` that provides a short rationale about the Cuda-coding exercises and presents the code structure; **make sure to read it**.
- Folder `histo-L3-thrashing` provides the code base for the first coding exercise referring to optimizing last-level cache (LLC) threshing in the context of a histogram-like computation.

- Folder `gpu-coalescing` provides the code base for the second coding exercise referring to optimizing GPU spatial locality (coalescing) by means of transposition.
- Folder `mmm` provides the code base for the third coding exercise referring to optimizing temporal locality for matrix-matrix multiplication.
- Folder `batch-mmm` provide the code base for the fourth coding exercise referring to optimizing temporal locality for a batch instance of matrix multiplication, in which the same matrices are multiplied but under a different mask.

Write a neat and short report containing the solutions to the first two theoretical questions, and also the missing code and short explanations for the four coding exercises in Cuda. Also provide comments regarding the performance behavior of your programs:

- what is the performance (in GB/sec or Gflops) and what is the speedup generated by your improvement with respect to the best performing GPU version provided?
- short and human-understandable rationale for justifying the speedup, for example:
  - for `histo-L3-thrashing` and `gpu-coalescing`: why does the optimized GPU program is several times faster than the GPU baseline in spite of performing a factor of 4x and 3x more accesses to global memory than the baseline, respectively?
  - for matrix multiplication and batch matrix multiplication: how does the optimization improves the temporal reuse?

1 Task 1: Pen and Paper Exercise Aimed at Applying Dependency-Analysis Transformations (3 pts)

Consider the C-like pseudocode below:

```
float A[2*M];

for (int i = 0; i < N; i++) {
    A[0] = i+1;

    for (int k = 1; k < 2*M; k++) {
        A[k] = sqrt(A[k-1] * i * k);
    }

    for (int j = 0; j < M; j++) {
        B[i+1, j+1] = B[i, j] * A[2*j  ];
        C[i,   j+1] = C[i, j] * A[2*j+1];
    }
}
```

Your task is to apply privatization, array expansion, loop distribution and loop interchange in order to parallelize as many loops as possible. Answer the following in your report:

- Explain why in the (original) code above **neither** the outer loop (of index `i`) **nor** the inner loops (of indices `k` and `j`) **are parallel**;

  to answer this question, we can visualize the dependency for the two loop nests.

  for i and k, we have that (i, k) reads from (k-1) in A, and thus, we cannot split up this computation, as we have a backward dependency.

  for i, j we have that (i, j) reads from (i-1, j-1) in B, and by the same reasoning.

- Explain why is it safe to privatize array `A`;

  we can privatize A as we have a false cross-iteration Write after Write(WaW) dependency in the k loop. this is true by the argument seen on slide 16/37 in loopparl5.pdf

- Once privatized, please explain "informally" why is it safe to distribute the outermost loop across the `A[0] = i+1;` statement and across the other two inner loops.

  After privatization of array `A`, it is safe to distribute the outermost loop because:

- The statement `A[0] = i+1;` only depends on the loop variable `i` and does not interact with other iterations.
- The first inner loop (index `k`) operates on the privatized array `A` independently for each iteration of `i`.
- The second inner loop (index `j`) uses values from `A` but does not modify it, and its operations on `B` and `C` depend only on the current `i` value.

Therefore, there are no loop-carried dependencies between these parts after privatization, making it safe to distribute the loop.

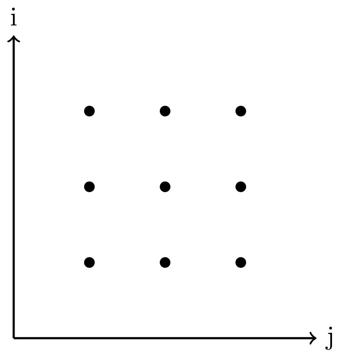
The distributed code with array expansion for `A` would look like this:

```
float A[N][2*M];   // Array expansion

for (int i = 0; i < N; i++) {
    A[i][0] = i+1;
}

for (int i = 0; i < N; i++) {
    for (int k = 1; k < 2*M; k++) {
        A[i][k] = sqrt(A[i][k-1] * i * k);
    }
}

for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        B[i+1, j+1] = B[i, j] * A[i][2*j   ];
        C[i,   j+1] = C[i, j] * A[i][2*j+1];
    }
}
```

- On the code resulted from the previous step, please reason in terms of direction vectors to determine which loops in each of (the resulted) three loop nests are parallel. Please explain why and annotate each loop with the comment `// parallel` or `// sequential`.

  loop 1 is parallel as we are just writing to non overlapping memory locations in a This would give us dependency matrix: [=, =] which can be parallelized.

Leftmost Loop



```
//parallel
for (int i = 0; i < N; i++) {
    A[i][0] = i+1;
}
```

for the second loop each iteration of k depends on an earlier iteration of k.

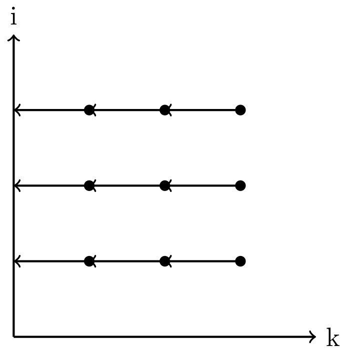
we have the following system of equations to solve:

(i1, k1) = (i2, k2-1) which gives us the direction vector [=, ¡]

using the theorem from slide 10/37: 'Th. Parallelism: A loop in a loop nest is parallel iff all its directions are either = or there exists an outer loop whose corresp. direction is ¡'

the outer loop i is parallel, but the inner loop over k, as it has '¡' direction and its outer loop does not have a '¡' direction, cannot be parallelized.

Leftmost Loop



```
//parallel
for (int i = 0; i < N; i++) {
    // sequential
```

```
        for (int k = 1; k < 2*M; k++) {
            A[i][k] = sqrt(A[i][k-1] * i * k);
        }
    }
```

for the last loop, we have

we have the following system of equations to solve:

for B (i1 + 1, j1 + 1) = (i2, j2) which gives us the direction vector [<, <]
for C (a1, b1 + 1) = (a2, b2) which gives us the direction vector [=, <]

by the same reasoning as for the second loop, the read write to C cannot be parallelized on both levels but only the outer loop. B can be parallelized on both levels and thus we could split it up into two

```
    // parallel
    for (int i = 0; i < N; i++) {
        // parallel
        for (int j = 0; j < M; j++) {
            B[i+1, j+1] = B[i, j] * A[i][2*j   ];
        }

        //sequential
        for (int j = 0; j < M; j++) {
            C[i,    j+1] = C[i, j] * A[i][2*j+1];
        }

    }
```

- Explain in terms of direction-vectors why is it legal to apply loop interchange on the third (last) loop nest. After interchange, please re-analyze the parallelism of the two loops in the third nest; has something changed?

  using the theorem from slide 12/37: 'Th. Loop Interchange: A column permutation of the loops in a loop nest is legal iff permuting the direction matrix in the same way does NOT result in a ¿ direction as the leftmost non-= direction in a row.'

  as interchange would imply [<, <] -¿ [<, <] for B and [=, <] -¿ [<, =] for C, loop interchange is legal.

  our code now looks like this:

```
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            B[i+1, j+1] = B[i, j] * A[i][2*j   ];
            C[i,    j+1] = C[i, j] * A[i][2*j+1];
        }
    }
```

Please show the code after interchange and annotate each loop with the comment `// parallel` or `// sequential`.

2 Task 2: Pen and Paper Exercise Aimed at Recognizing Parallel Operators (3 pts)

Assume that both A and B are matrices with N rows and 64 columns. Consider the pseudocode below:

```
float A[N,64];
float B[N,64];
float accum, tmpA;
for (int i = 0; i < N; i++) { // outer loop
    accum = 0;
    for (int j = 0; j < 64; j++) { // inner loop
        tmpA = A[i, j];
        accum = sqrt(accum) + tmpA*tmpA; // (**)
        B[i,j] = accum;
    }
}
```

Reason about the loop-level parallelism of the code above and answer the following in your report:

1. Why is the outer loop **not** parallel?

   the loop is not parallel as all the threads will write to the same accum variable, and thus, we will have a race condition.

2. Please explain what technique can be used to make it parallel and why is it safe to apply it? Re-write the code such that the outer loop is parallel, i.e., the outer loop does not carry any dependencies.

   we can use privatization of accum to remove inter iteration dependencies.

   ```
   float accum, tmpA;
   #pragma omp parallel for private(i, accum)
   for (int i = 0; i < N; i++) { // outer loop
       accum = 0;
       for (int j = 0; j < 64; j++) { // inner loop
           tmpA = A[i, j];
           accum = sqrt(accum) + tmpA*tmpA; // (**)
           B[i,j] = accum;
       }
   }
   ```

3. Explain why the inner loop is **not** parallel.

   the inner loop is not parallel for much the same reason as the outer loop, just with respect to a different variable tmpA.

4. Assume the line marked with `(**)` is re-written as `accum = accum + tmpA*tmpA`. Now it is possible to rewrite both the inner and the outer loop as a nested composition of parallel operators! Please write in your report a semantically-equivalent, nested-parallel Futhark program.

```
        float accum, tmpA;
        for (int i = 0; i < N; i++) { // outer loop
            accum = 0;
            for (int j = 0; j < 64; j++) { // inner loop
                tmpA = A[i, j];
                accum = accum  + tmpA*tmpA; //
                B[i,j] = accum;
            }
    }
```

   in futhark this can be rewritten as composition of a map and a scan IE a segmented scan.

   the code would look like this:

```
    let bmm [n] (A: [n][64]i64) : [n][64]i64 =
        map (\row ->
            let squares = map (\x -> x*x) row
            let scanned = scan (+) 0 squares
            in scanned
        ) A


    -- compiled input { 30i64 }
    entry main (N: i64) : [][]i64 =
    let A : [N][64]i64 = replicate N (iota 64)
    let B = bmm A
    in B
```

3 Task 3: Histogram-like Computation – Cuda Exercise 1 (3 pts)

See section "LL$ threshing: Histogram-like computation" in companion lecture slides `L6-locality.pdf`.

The programming task refers to implementing the missing code in files `main-gpu.cu` and `kernels.cu.h`—search for keyword "Exercise" in those files and follow the instructions.

Program arguments are, e.g., see Makefile:

- The first argument of the program is the size `N` of the array of indices/-values.

- The second argument of the program is the size of the last-level cache (LL$) in bytes. Please make sure to adjust it to the hardware you are running on (both CPU and GPU), otherwise you will not observe much. The sizes used in the makefile are particularized to the `hendrixfut01fl` and `hendrixfut03fl` machines.
- The size of the histogram is computed internally such as four passes over the input are always performed.

Briefly comment in your report on:

- the code implementing your solution, i.e., present the code and comment on its correctness and on how it optimizes locality. For example, why do you expect speedup when the improved implementation performs a factor of 3-4x more accesses to global memory (since it traverses the input four times).
- specify whether your implementation validates
- report the GB/sec achieved by your implementations and of the GPU baseline and also report the speedup in comparison with the GPU baseline (i.e., the other provided implementation)

solution code:

```
if(gid < N) {
    uint32_t ind = inp_inds[gid];

    if(ind < UB && ind >= LB) {
        float val = inp_vals[gid];
        atomicAdd((float*)&hist[ind], val);
    }
}
```

//TODO, better explanation the reason that allthough we are doing more accesses to global memory, is that we add global memory accesses but improve the coalsced access in shared memory, which turns out to be more important.

The implementation validates and we get following performance results with evaluating on an A100 40GB gpu

Implementation	**Runtime (ups)**	**GB/sec**	**Speedup**
CPU Original	1,776,744	4.96	-
CPU Multi-Pass	1,049,537	8.40	1.69x
GPU Original (Baseline)	73,155	118.40	-
GPU Multi-Step (Optimized)	19,127	452.86	3.82x

Table 1: Performance comparison of histogram implementations

The multi-step kernel achieves better performance by reducing cache thrashing. By processing the histogram in smaller chunks (LB to UB), it ensures that each chunk fits within the GPU's last-level cache. This significantly reduces cache misses and improves memory access efficiency. Although it performs more global memory accesses overall, the improved cache utilization and reduced memory latency outweigh this cost

4 Task 4: Optimizing Spatial Locality by Transposition – CUDA exercise 2 (4 pts)

See section "Optimizing Spatial Locality by Transposition" in companion lecture slides `L6-locality.pdf`.

The programming task refers to implementing in folder `gpu-coalescing`:

1. In file `goldeSeq.h`, please **correctly** parallelize by means of OpenMP the outer loop `for(uint64_t i = 0; i < num_rows; i++) ....`
2. The code of CUDA kernel `transKernel` in file `kernels.cu.h`, which works on the transposed versions of A and B, named `A_tr` and `B_tr`, respectively. Please search for keyword "Exercise" in file `kernels.cu.h` to find the implementation place.

Please include in your report:

- The OpenMP (parallel) code of `goldeSeq.h`; report the speedup obtained by parallelizing golden sequential, i.e., sequential CPU runtime divided by parallel runtime.

```
#pragma omp parallel for private(a_el, accum)
for(uint64_t i = 0; i < num_rows; i++) {
    uint64_t ii = i*num_cols;
    accum = 0.0;
    for(uint64_t j = 0; j < num_cols; j++) {
        a_el  = A[ii + j];
        accum = sqrt(accum) + a_el*a_el;
        B[ii + j] = accum;
    }
}
```

  we get the following performance results:

Implementation	**Runtime (us)**	**GB/sec**	**Speedup**
CPU Original	343043	1.565025	-
CPU parallel	6811	78.824097	50x

Table 2: Performance comparison of baseline and optimized implementations

- the CUDA-kernel code implementing your solution, i.e., present the code and comment on its correctness and on how it optimizes spatial locality (i.e., coalesced access to global memory). For example, why do you expect speedup when **your** implementation performs a factor of 3x more access to global memory than the baseline.

  the cuda implementation of the transpose kernel VALIDATES and is as follows:

  ```
  template<class ElTp>
  __global__ void
  transKernel(ElTp* A_tr, ElTp* B_tr, uint32_t num_rows, uint32_t num_cols
      uint32_t gid = blockIdx.x * blockDim.x + threadIdx.x;
      if(gid >= num_rows) return;

      ElTp accum = 0;
      for(int j=0; j<num_cols; j++) {
          ElTp el_a  = A_tr[j*num_rows + gid];
          accum = sqrt(accum) + el_a * el_a;
          B_tr[j*num_rows + gid] = accum;
      }
  }
  ```

  1. **Correctness:** The kernel maintains the same computation logic as the naive version, but operates on transposed matrices. The accumulation and result calculation remain identical, ensuring correctness.
  2. **Coalesced memory access:** In the optimized kernel, consecutive threads access consecutive memory locations (e.g., `A_tr[j*num_rows + gid]`), resulting in coalesced memory access. This is in contrast to the naive kernel where threads access memory with stride `num_cols`.
  3. **Improved performance despite more memory accesses:** While the optimized version performs 3x more global memory accesses due to the initial transposition, the benefit of coalesced access far outweighs this cost. Coalesced access allows for efficient use of memory bandwidth, reducing the overall memory access latency and improving throughput.
  4. **Better cache utilization:** The coalesced access pattern leads to better utilization of the L1 and L2 caches, as each cache line fetched contains useful data for multiple threads.

  performance results:

- briefly explain why the CPU implementation that uses GPU-like coalescing has abysmal performance (i.e., much slower than the baseline).

  Program with GPU-like coalescing runs on CPU in: 696606 microsecs, GB/sec: 0.770695 This performance is abysmal due to the fundamental

Implementation	Runtime (us)	GB/sec	Latency Speedup	Throughput Speedup
GPU Baseline	7377	72.78	-	-
GPU Optimized	1332	403.06	5.54x	5.54x

Table 3: Performance comparison of baseline and optimized implementations

difference in how memory is designed to be used on gpus vs cpus. On gpus, having consecutive threads in a block access consecutive memory is optimal, as this is coalesced, On the other hand this is not optimal for cpus, as they rely heavily on caching for performance. When using GPU-like coalescing on a CPU, each thread accesses memory that is far apart, leading to poor cache line utilization. That is, the same approach that creates locality on the gpu destroys locality on the cpu

- **BONUS** briefly explain at a very high level, why/how "the Optimal-GPU Program" is about 2x faster than your implementation.

  The "Optimal-GPU Program" achieves about 2x speedup over the transposed implementation primarily due to its more efficient use of GPU resources. While both kernels ensure coalesced memory access, the optimal version goes a step further by utilizing shared memory as a cache. It processes data in chunks (`CHUNK`), reducing global memory transactions significantly:

  ```
  for(int jj=0; jj<num_cols; jj+=CHUNK) {
      // Load chunk to shared memory
      // Process data in shared memory
      // Write results back to global memory
  }
  ```

  This approach contrasts with the transposed kernel's direct global memory access for each element:

  ```
  for(int j=0; j<num_cols; j++) {
      ElTp el_a  = A_tr[j*num_rows + gid];
      // Process and write back to global memory
  }
  ```

  The optimal kernel's chunked processing also allows for better instruction-level parallelism and reduces potential thread divergence. By leveraging shared memory and minimizing global memory accesses, the optimal kernel makes more efficient use of the GPU's memory bandwidth, leading to the observed performance improvement.

  ("the Optimal-GPU Program" is the last GPU program run by the Makefile)

5 Task 5: Matrix-Matrix Multiplication (MMM) – Cuda Exercise 3 (4 pts)

See section "L1$ and Register: Matrix-Matrix Multiplication" in companion lecture slides `L6-locality.pdf`.

The programming task refers to implementing in folder `mmm` some of the code of Cuda kernel `mmmSymBlkRegInnSeqKer` in file `kernels.cu.h`. Please search for keyword "Exercise" in file `kernels.cu.h` to find the implementation place, and follow the instructions there. Also look around to see how it is called from the CPU (host) code.

Please be aware that Section 6.4 of lecture notes presents a different tiling strategy for matrix-matrix multiplication; i.e., it is related but it is **not** what you have to do.

Briefly comment in your report on:

- the code implementing your solution, i.e., show the code and comment on it, e.g., explaining why the access to global memory is coalesced

  the code is as follows:

```
for(int kk = 0; kk < widthA; kk += Tk) {
    for (int i = 0; i < Ry; i++) {
        int row = iii + threadIdx.y * Ry + i;
        int col = kk + threadIdx.x;
        if (row < heightA && col < widthA) {
            Aloc[threadIdx.y * Ry + i][threadIdx.x] = A[row * widthA + c
        } else {
            Aloc[threadIdx.y * Ry + i][threadIdx.x] = 0.0;
        }
    }

    for (int j = 0; j < Rx; j++) {
        int row = kk + threadIdx.y;
        int col = jjj + threadIdx.x * Rx + j;
        if (row < widthA && col < widthB) {
            Bloc[threadIdx.y][threadIdx.x * Rx + j] = B[row * widthB + c
        } else {
            Bloc[threadIdx.y][threadIdx.x * Rx + j] = 0.0;
        }
    }

    __syncthreads();

    // compute the per-thread result css:
    for(int k = 0; k < Tk; k++) {
        #pragma unroll
        for(int i=0; i<Ry; i++) {
                #pragma unroll
                for(int j=0; j<Rx; j++) {
                    css[i][j] +=
                      Aloc[threadIdx.y * Ry + i][k] *
                      Bloc[k][threadIdx.x * Rx + j] ;
                }
            }
        }
    }
```

For matrix A, the code uses `threadIdx.x` to access consecutive columns within each row, ensuring that adjacent threads read contiguous memory locations. Similarly, for matrix B, the code uses `threadIdx.x * Rx + j` to access consecutive elements within each row, maintaining coalesced access.

- report the performance in Gflops achieved by **your** GPU implementation and by the GPU **baseline** , and also report the speedup w.r.t. the baseline.

Implementation	**Runtime (us)**	**GFlops/sec**	**Speedup**
Naive Baseline	14279	2406.31	-
Optimized	2859	12018.10	5

Table 4: Performance comparison of baseline and optimized implementations

- Finally, explain in your report the high-level reasons for obtaining this speedup, i.e., how did your implementation improved the temporal locality (e.g., by what factor has decreased the number of accesses to global memory).

  the speedup is mainly achieved by preloading data that will be reused in the computation into shared memory instead of accessing global memory for every memory access.

  In the naive implementation, assuming N = heightA = widthA = widthB, we are reading from global memory $2N^3$ elements (N^3 reads each for matrices A and B).

  In our tiled implementation, with tile dimensions $T = T_x = T_y = T_k$, and $R = R_x = R_y$:

  1. We load tiles of size $T \times T \times R$ for both A and B into shared memory.
  2. These tiles are reused T times within each block.
  3. We need $(N/T)^3$ such tile loads to cover the entire computation.

  So, the number of global memory reads in the tiled version is:

$$2 \cdot (N/T)^3 \cdot T \cdot T \cdot R = 2N^3R/T$$

Comparing this to the naive version:

$$\text{Reduction factor} = \frac{2N^3}{2N^3R/T} = T/R$$

With $T = 16$ and $R = 4$ in our implementation, we get a reduction factor of $16/4 = 4$.

This means we've reduced global memory accesses by a factor of 4, which significantly improves temporal locality and contributes to the observed speedup.

Additionally, the use of shared memory allows for faster access to the preloaded data, further improving performance. The tiled approach also enables better use of the GPU's parallel processing capabilities by allowing multiple thread blocks to work concurrently on different tiles.

6 Task 6: Batched Matrix Multiplication Under a Mask – Cuda Exercise 4 (4 pts)

See section "L1$ and Register: Batch Matrix Multiplication under a Mask" in companion lecture slides `L6-locality.pdf`.

The programming task refers to implementing in folder `batch-mmm` the code of the Cuda kernel `bmmmTiledKer` in file `kernels.cu.h`. Please search for keyword "Exercise" in file `kernels.cu.h` to find the implementation place, and follow the instructions there. Remember to flatten the indices to all arrays hold in global memory. Also look around to see how it is called from the CPU (host) code.

Briefly comment in your report on:

- the code implementing your solution,

  the code(which validates) is as follows:

```
template <class ElTp, int T> __global__
void bmmmTiledKer (
ElTp* A,
ElTp* B,
char* X_tr,
ElTp* Y,
const int M,
const int K,
const int N
) {
    __shared__ ElTp Xsh_tr[T];
    ElTp acc[T];

    const int ii  = blockIdx.x;
const int j1  = threadIdx.y;
const int j2  = threadIdx.x;
const int i   = ii * T;
const int flat_idx = threadIdx.y * K + threadIdx.x;

#pragma unroll
for(int t=0; t<T; t++)
    acc[t] = 0;

/*********************************************
*** Cuda Exercise 4: ***
*
* With the help of the pseudocode from the
* lecture slides, please implement the rest of
* the code of this kernel.
* Remember to flatten the indices to all arrays
* hold in global memory, i.e., A, B, X_tr, Y.
*********************************************/

// Main computation loop
for (int q = 0; q < N; q++) {
    ElTp a = A[j1 * N + q];
    ElTp b = B[q * K + j2];
    // Update shared memory for next iteration
    char tmp = 0;
    if (flat_idx < T && i + flat_idx < M) {
    tmp = X_tr[q * M + i + flat_idx];
    }
    Xsh_tr[flat_idx] = tmp;

    __syncthreads();

    #pragma unroll
    for (int i_r = 0; i_r < T; i_r++) {
    if (i + i_r < M) {
        ElTp x = (Xsh_tr[i_r] != 0) ? 1.0 : 0.0;
        acc[i_r] += a * b * x;
    }
    }
    __syncthreads();
}

// Write results back to global memory
#pragma unroll
for (int i_r = 0; i_r < T; i_r++) {
    if (i + i_r < M) {
            Y[(i + i_r) * K * K + j1 * K + j2] = acc[i_r];
        }
    }
}
```

- report the performance in Gflops achieved by **your** GPU implementation

Implementation	**Runtime (us)**	**GFlops/sec**	**Speedup**
Naive Baseline	5894	2184.90	-
Optimized	2334.5	5516.36	2.53

Table 5: Performance comparison of baseline and optimized implementations

- Finally, explain in your report the high-level reasons for obtaining this speedup, i.e., how did your implementation improved the temporal locality (e.g., by what factor has decreased the number of accesses to global memory).

  The speedup is primarily achieved through improved temporal locality and memory coalescing and reduced global memory accesses. By using shared memory to store a tile of the X matrix (size T=31), we significantly decrease the number of global memory reads.

  Total accesses for naive implementation: $O(M \times K^2 \times N)$

  for the optimized implementation:

  - Accesses to X_tr: Reduced from $O(M \times K^2 \times N)$ to $O(M \times N)$
  - Accesses to A and B: Remain $O(M \times K^2 \times N)$

  Reduction factor for X_tr accesses:

$$\frac{O(M \times K^2 \times N)}{O(M \times N)} = O(K^2)$$

  thus we have a reduction by a factor of $O(K^2)$ in global memory accesses for X_tr, while accesses to A and B remain unchanged