Final project

MATH 471/571 - Parallel Scientific Computing

Michal A. Kopera

Due date: May 5th, 2021

Note there is no second submission date. If you want my feedback, reach out to me ahead of time and we can talk about your project.

You will choose **one** problem and work on the tasks outlined there. Submit a PDF write-up along with the code and other files (Makefile, run script) you have used to obtain the results. You submit your solution by putting it in a folder named Final Project in your Google Drive personal folder. Please refer to the Syllabus regarding how the final project affects your final grade.

The projects are your individual work. I am happy to clarify some muddy points, but I would like to see your individual attempt to solve the project. If you got help from somebody, or used a resource outside class (i.e. website) please provide appropriate acknowledgement and/or reference. **You can** discuss your ideas on Slack, but **you cannot** share your codes.

Grading policy:

This is a slight departure from the policy outlined in the Syllabus, regarding the final project, where I specify the percentage of points required to get to earn each grade. Instead of awarding you points out of 100%, each problem has three tasks. If you are aiming at C, complete Task 1 only. If you want to get a B, complete tasks 1 and 2. To earn a grade A, you have to complete all the tasks. There is no separate mastery question. To earn each grade you have to have appropriate number of project points, and sufficient quiz scores.

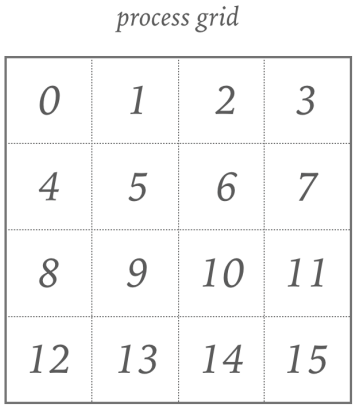
Problem 1 - Matrix multiplication using MPI and CUDA

In Project 3 you have examined different options for writing CUDA code for matrix-matrix multiplication. In this project, you will explore this problem further, and write an MPI code for matrix-matrix multiplication. You will compare the MPI code to your CUDA code, and decide which approach uses the resources more efficiently.

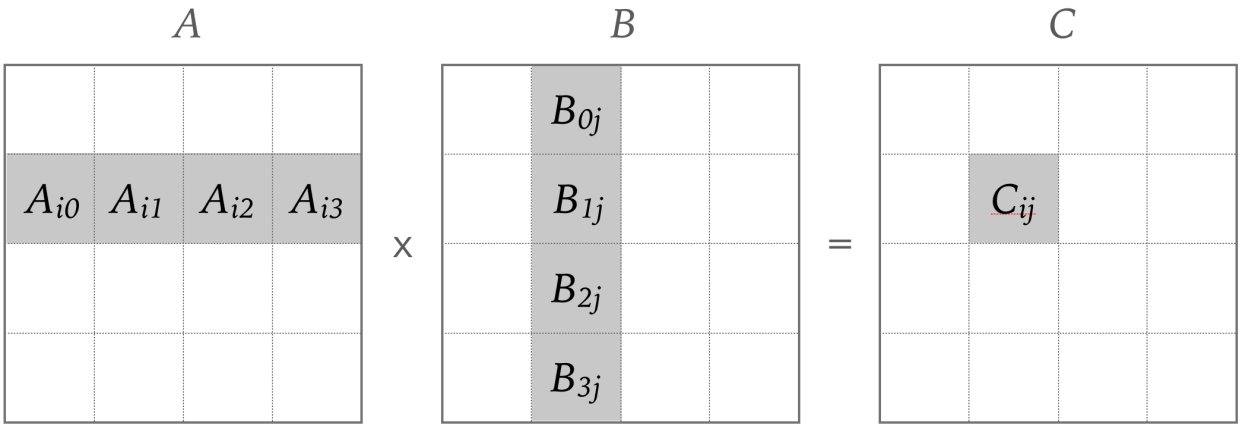
Matrix-matrix multiplications looks like a perfect candidate for parallelization, as each element of the resulting matrix C is computed independently of the others:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}.$$

In the CUDA implementation, you have likely concluded that a 2D-2D decomposition, where you divided matrix C in a 2D grid of blocks, and each block in a 2D grid of threads, was the fastest solution. Let's try to apply a similar idea in the distributed memory setting, where we need to distribute matrices A, B and C among the ranks, where each rank has access only to it's own fragment of each matrix.



Assume we have p processes (ranks), and p is a square number (i.e. $q = \sqrt{p}$ is an integer), we want to distribute A, B, C among processes such that each process owns a sub-matrix $A_{i,j}$, $B_{i,j}$, $C_{i,j}$ of size $n/q \times n/q$, and processes are organized in an $q \times q$ grid with the coordinates (i, j) representing the location of the process in the grid, and the global rank of the process (i, j) can be



obtained as $r = i * q + j$. In the example depicted here we use $p = 16$, which leads to $q = 4$. To compute sub-matrix $C_{i,j}$ we then need not only $A_{i,j}$ and $B_{i,j}$, but also the

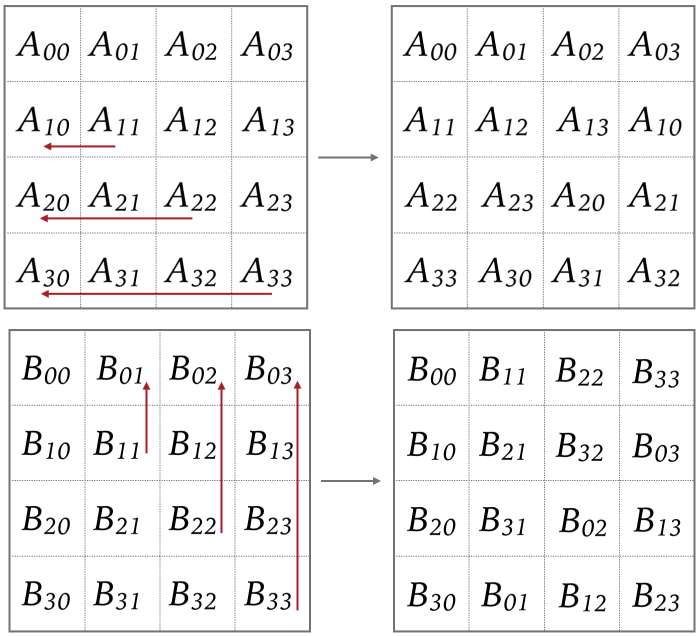
entire row of sub-matrices $A_{i,0},\ A_{i,1},\ \ldots\ ,\ A_{i,q-1}$, and the entire column of sub-matrices $B_{0,j},\ B_{1,j},\ \ldots\ ,\ B_{q-1,j}$

$$C_{i,j} = A_{i,0}B_{0,j}\ +\ A_{i,1}B_{1,j}\ +\ \ldots\ +\ A_{i,q-1}B_{q-1,j}.$$

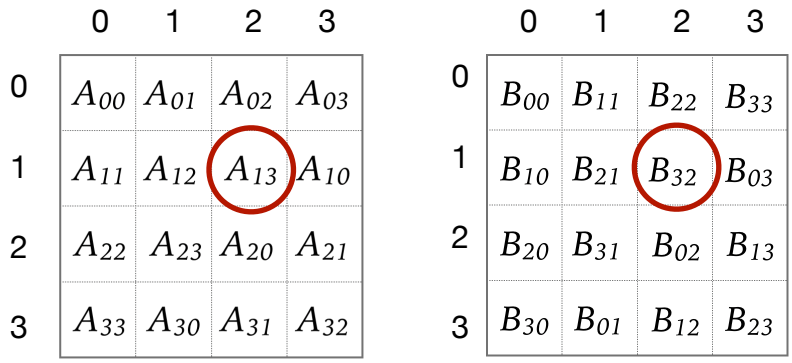
The naive algorithm would then require to gather n/q rows of A and n/q columns of B onto rank (i,j), and then compute $C_{i,j}$. This would require us to store a total of $2q+1$ sub-matrices of size $n/q \times n/q$ on each process, for a total of $(2q+1)\cdot\frac{n}{q}\cdot\frac{n}{q} = \frac{(2q+1)n^2}{q^2} \approx \frac{2n^2}{\sqrt{p}}$ numbers. The total memory required for all processes is then approximately $\frac{2n^2}{\sqrt{p}}\cdot p = 2n^2\sqrt{p}$, which is considerably more than the serial algorithm memory footprint of $3n^2$ (the space needed to store three $n \times n$ matrices). To alleviate that problem, a Canon algorithm was devised, which assumes that we only store a single $A_{i,j},\ B_{i,j},\ C_{i,j}$ on each process (i,j), leading to a per-process memory requirement of $3n^2/p$, and $3n^2$ for all processes. The Canon algorithm is executed in the following steps:

1) Initially, process (i,j) owns sub-matrices $A_{i,j},\ B_{i,j},\ C_{i,j}$

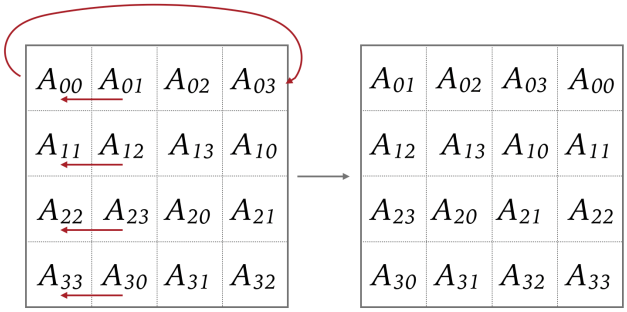
2) We perform initial alignment of sub-matrices within the process array by sending each block $A_{i,j}$ to process $\left(i,\ (j-i+q)\mod q\right)$, and each block $B_{i,j}$ to process $\left((i-j+q)\mod q,\ j\right)$ - see image below (left is before shift, right is after)

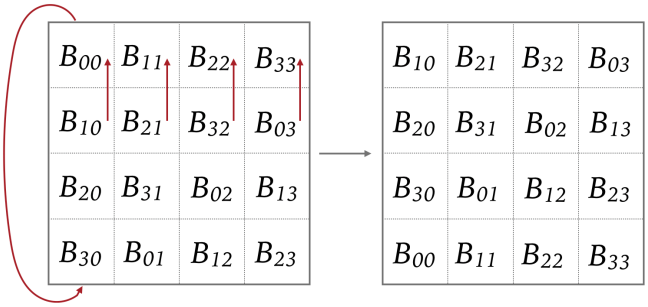


3) Each rank multiplies its local A and B sub-matrix, and increments its local $C_{i,j} += A_{loc}B_{loc}$. For example, $C_{1,2} += A_{1,3}B_{3,2}$.



4) Each local sub-matrix A is shifted to the neighbor directly to the left, i.e. $(i, j) \rightarrow (i, j-1)$, and each sub-matrix B is shifted to the neighbor directly above, i.e. $(i, j) \rightarrow (i-1, j)$. This shift assumes periodicity, so the left-most process sends its A to the right-most process in the same row, and top-most process sends its B to the bottom-most process.





5) Compute $C_{i,j} += A_{loc}B_{loc}$, where A_{loc}, B_{loc} are currently locally stored sub-matrices by process (i, j). For example, after the first shift $C_{1,2} += A_{1,0}B_{0,2}$

6) Repeat steps 4 and 5 for a total of $q-1$ times, until the complete $C_{i,j}$ is computed:

$$C_{i,j} = A_{i,0}B_{0,j} \; + \; A_{i,1}B_{1,j} \; + \; \ldots \; + \; A_{i,q-1}B_{q-1,j}$$

Task 1

Implement Cannon algorithm as outlined above. To compute local matrix-matrix product, use a serial algorithm from Project 3. Show that your parallel algorithm produces a correct result by testing it on a small problem using 4 processors and comparing it to the serial computation of the multiplication of the same matrices. Use the code in Project 3 to generate random matrices to multiply.

Task 2

Design an experiment, where you multiply two big matrices (you can use $N > 2^{11}$, since we know CUDA can multiply this size very quickly) using your best thread and block setting in CUDA using your code from Project 3, and the MPI code from Task 1 with $p = 4, 16, 64$. Time the execution of both CUDA code (including memory transfers) and the MPI code (including all communication). Present the results in a table or a plot. What are your conclusions from this comparison? If the CUDA code is faster, how many processes would you need with the MPI code to match the performance of CUDA code?

Task 3

The CUDA code from Project 3 uses only global memory. There is a room for improvement if we use shared memory for that code. Note that in your 2D-2D implementation a thread which computes $C_{i,j}$ has to access the entire i-th row of matrix A, and the entire j-th column of matrix B. The thread which computes $C_{i,j+1}$, will again access the same i-th row of matrix A, and j+1st column of matrix B. This means that different threads access the same locations in global memory. We could save some time by loading values of A and B to shared memory, and thus make reusing those values by different threads much faster. Because the amount of shared memory is limited, and we cannot dynamically allocate it, we can utilize some of the ideas that stem from the Cannon's algorithm above. Consider the following algorithm:

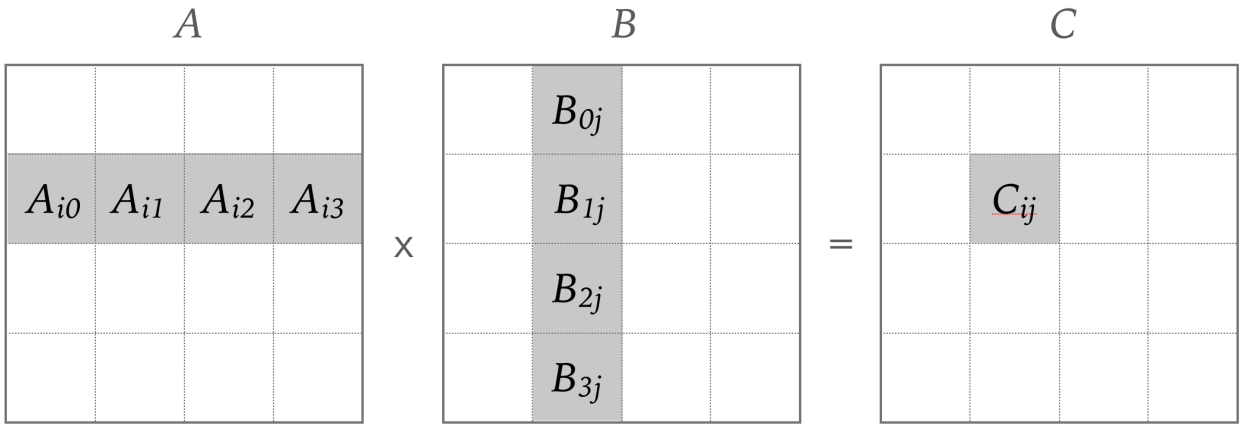
1) Assume we use blocks which are 32x32 threads big (you can use a different value here). Allocate shared memory arrays A_shared, B_shared, C_shared which are 32x32 size. For the purpose of this example, the C_{ij} in the figure below symbolizes a tile of matrix C assigned to block i, j in the 2D grid decomposition. The tile is 32x32 large.

2) Load tile A_{i0} to A_shared and B_{0j} to B_shared such that each thread loads one value of tile A and B to appropriate location in shared memory. Make sure you synchronize threads after you load the shared memory

3) Each thread in a block performs matrix multiplication $C_{ij} += A_{i0}B_{0j}$ and updates the C_shared location. Note that in the above notation A_{i0} stands for a tile of matrix A corresponding to block with coordinated (i,0), so each thread needs to multiply the appropriate row of the tile A_{i0} with a column of tile B_{0j}.

4) Each thread loads another tile of matrices A and B (A_{i1} and $B_{1,j}$) and updates the location in C_shared with the result of $C_{ij} += A_{i,1}B_{1,j}$. This step is repeated until we load all the tiles and complete the computation.

5) Once the loop over tiles is finished, each thread writes the result from C_shared to global memory.



Once you implement this algorithm and verify its correctness, time it and compare against the old CUDA code, and the MPI result you got in Task 2. What are your conclusions from this study?

Hints:

1) If this algorithm seems too complicated, try to first use shared memory for C array only. This should speed up the computation by a bit, as we have to repeatedly read and write to/from C in a loop executed by each thread.

Optional ideas:

1) If you want to push the performance of your CUDA kernel, you may want to exploit something called a coalesced memory access. We did not discuss it in class, but this is a pretty simple idea that it is faster to access memory locations which are close to each other, rather than spread apart. For instance, if I store my matrix as a one long vector such that rows are kept together, I can loop over each row and access elements much faster, than if I looped over a column. This is because the

computer (and GPU) typically loads memory locations in chunks, rather than one-by-one. Unfortunately, in the matrix-matrix multiplication we have a loop over both rows and columns. However, since we load matrix tiles into shared memory, we could transpose tile of matrix B such that the columns are now rows, and modify the matrix multiplication kernel such that we multiply elements of rows of tile of matrix A, with elements of rows of the transposed tile of matrix B. This transposition is basically free, since we copy data from one memory to the other, it is just a matter of modifying the indices, and making sure the kernel loop is modified appropriately. It should speed-up your kernel significantly, at a price of a slightly less clean code - but this is typically the price we pay for performance.

Problem 2 - Reaction-diffusion equation

You have worked with the infamous MPI implementation of the Barkley model in Project 2. Here I give you a chance to revisit this problem using CUDA.

Task 1 - Barkley model with CUDA

Use the serial code from Project 2 and write a CUDA version using global memory. Show that you get the same result as the serial code by presenting the side-by-side plots of the solution (either u or v variables). Use the same problem set-up as in Project 2. Use what you have learned in Project 3 to choose the decomposition of blocks and threads.

Task 2 - Performance comparison

Time your CUDA code using various settings of threads per block (i.e., if you use 2D-2D decomposition, it could be 4x4, 8x8, 16x16, 32x32 threads per block). Time the entire code, including kernels and memory transfers.

Compare those times against the times you obtained for the MPI code. To keep things simple, use the problem size which is a power of 2 (i.e. 2^{11}), and square powers of 2 for number of processors (i.e. 4, 16, 64). You can present the data as a table, or a plot. Write your conclusions from that comparison. If the CUDA code is faster, how many processes in the MPI code would you need to match the GPU performance?

Task 3 - Barkley model with shared memory

Since we perform multiple time-steps in the Barkley model computation, it is a good candidate for exploiting shared memory. Propose an algorithm which would use shared memory, implement it and time it in the same way as in Task 2. Compare the timing results to the results of Task 2. Comment on your findings.

Problem 3 - Monte-Carlo integration[1]

One of the applications of the statistical Monte-Carlo method is the numerical integration of complex integrals. In a nutshell, the Monte-Carlo method is like throwing darts and counting how many of them fits in the area under the curve. If you throw enough darts, this method will yield you a good approximation to the integral. Additionally, this method is also a perfect example of a parallel algorithm as each dart is completely independent of each other, so the algorithm is (mostly) completely parallel. The only challenge is to aggregate the results together, which comes down to a reduction operation. In this project you will explore the Monte Carlo method.

Background on Monte Carlo method

Suppose $p(x)$ is a probability distribution function, and we wish to calculate the integral

$$\int_{-\infty}^{\infty} p(x)f(x)dx.$$

To estimate that integral, take N random samples $x_1, \ldots, x_N$ from the distribution given by $p(x)$; then estimate of the integral is given by:

$$\int_{-\infty}^{\infty} p(x)f(x)dx = \langle f(x)\rangle \approx \frac{1}{N}\sum_{k=1}^{N} f(x_k).$$

The question remains, how big N has to be for the above expression to converge to a reasonable approximation? D.L. Chopp (or any statistics book including topics on Monte Carlo methods) gives a step-by step explanation using the Chebyshev inequality. For the purpose of this project we will assume that in order to be 99% confident that our estimation has reached the tolerance ϵ we will take:

$$N \geq \frac{100\,\mathrm{var}\{p(x)\}}{\epsilon^2},$$

where $\mathrm{var}\{p(x)\}$ is the variance of the distribution $p(x)$.

Let's take a look at an example. Say we use $p(x)$ which is a uniform distribution for the interval $[0, 1]$, which has a variance $\mathrm{var}\{p(x)\} = \frac{1}{12}$, and x_k are samples from that distribution. Then

$$\int_{-\infty}^{\infty} p(x)f(x)dx = \int_{0}^{1} f(x)dx \approx \frac{1}{N}\sum_{k=1}^{N} f(x_k).$$

---

[1] from D.L. Chopp, Introduction to High-Performance Scientific Computing

So for an error tolerance of $\epsilon = 10^{-2}$ and 99% confidence of meeting that tolerance, we need to take

$$N \geq \frac{100 \cdot \frac{1}{12}}{(10^{-2})^2} \approx 83333 \text{ samples.}$$

It is worth to notice that taking a very large number of samples (i.e. 10^{12}) may not be beneficial, since the roundoff error in the addition of many floating-point numbers will add up and decrease the accuracy. It is important to choose N such that it is large enough, per the criterion above, but not too large.

Problem description

Use Monte Carlo integration to estimate the integral

$$\int_0^\infty e^{-\lambda x} \cos x dx$$

for $\lambda > 0$ by using an exponential distribution $p(x) = \lambda e^{-\lambda x}$ for $x \geq 0$, which has a variance $\text{var}\{p(x)\} = \frac{1}{\lambda^2}$. Note that when using the $p(x)$ distribution, the integral has to be rewritten as

$$\int_0^\infty e^{-\lambda x} \cos x dx = \int_0^\infty \lambda e^{-\lambda x} \frac{\cos x}{\lambda} dx = \int_0^\infty p(x) \frac{\cos x}{\lambda} dx.$$

Thus, $f(x) = \frac{1}{\lambda} \cos x$.

To generate random values with distribution $p(x)$, let X be a random value obtained from a uniform distribution on the interval $[0, 1]$; then a random variable Y drawn from $p(x)$ can be obtained by transformation

$$Y = -\frac{1}{\lambda} \ln X.$$

For this problem, we know the exact value of the integral, which is

$$\int_0^\infty e^{-\lambda x} \cos x dx = \frac{\lambda}{1 + \lambda^2}.$$

Task 1 - Monte Carlo integration with MPI

Program the above problem using MPI. Your code should take N samples as an input parameter. Measure the time of the entire program using MPI_Wtime function, and report the elapsed time in seconds. Each rank should use a different random seed to

make sure it is generating different pseudo-random numbers. You can use the rank number as a seed, or come up with some other idea.

1) Use $\lambda = 1$, and determine a number of samples required to achieve error tolerance $\epsilon = 10^{-4}$ with confidence 99%. Run your code with this N to verify that your solution is indeed within the specified tolerance from the exact solution.
2) Measure the time required to compute a solution for $N = 10^q$ samples, where $q = 3, 4, \ldots, 10$ and plot the time versus N. Use the plot to estimate the time required to meet the error tolerance $\epsilon = 10^{-5}$.
3) Create both weak and strong scaling plots for $p = 1, 2, 4, 8, 16, 32, 64$ processors. Choose the problem size such that N is the same for strong and weak scaling for $p = 64$ processors.

Task 2 - Monte Carlo integration with CUDA

Program the Monte Carlo integration problem above using CUDA. The strategy could be as follows (but feel free to explore your own):

1) Each thread generates $N/(BT)$ random numbers (using unique random seed) from the distribution $p(x)$, where N is the problem size, B is the number of blocks you use, and T is the number of threads per block.
2) Each thread applies function $f(x)$ to the list of its random numbers, and computes the sum of results
3) The partial sums from each thread are reduced to a single total from all the threads. This could be done in a separate kernel to keep the program simple. This is a reduction operation, so be mindful of race conditions and use appropriate addition operation. The total should then be divided by N to get the estimate for the integral.

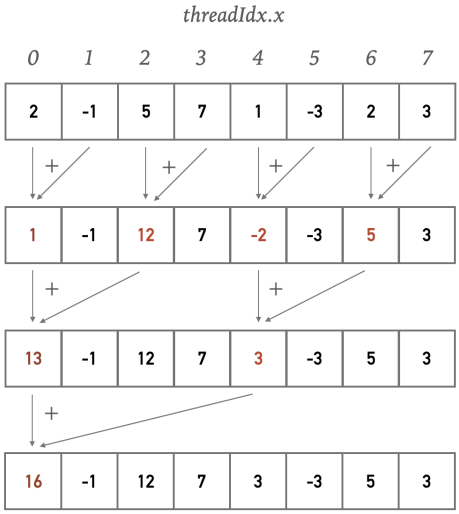
Similarly like in Task 1, first make sure your program gives you correct results for $\lambda = 1$, and N large enough to achieve error tolerance $\epsilon = 10^{-4}$ with confidence 99%.

Then, measure the time required to compute a solution for $N = 10^q$ samples, where $q = 3, 4, \ldots, 10$ and plot the time versus N. Compare this result with the MPI code in Task 1. Explain your choice of threads per block, and the number of blocks. You can only use a maximum of 65535 block per grid dimension, which is likely not enough to create enough threads to cover large N values.

Task 3 - Reduction operation in CUDA

In this problem, the key to achieving good performance with CUDA is to write an efficient reduction operation to sum up all the partial results. You have likely used atomicAdd operation in Task 2, but this is not the most efficient way of doing this, as all

the threads are essentially summing up the results one-by-one, thus serializing the operation. In this task you will explore the possibility of a more efficient reduction operation.



The reduction operation can be divided into two stages: 1) within a block, 2) between blocks. Within a block, we need to sum-up the results from each thread. The naive way of doing that is for one thread to sum-up all the entries, possibly using shared memory to speed up the process. But we can keep more threads occupied if we employ the following algorithm:

1) threads with even indices (i.e. threadIdx.x%2 == 0) sum up its own result and the result of a neighboring thread (i.e. threadIdx.x + 1), and save it back to shared memory array in it's own location
2) threads with indices threadIdx.x%4 ==0 sum up their own result (which is already a result of the addition in step 1), and that of a thread threadIdx.x+2.
3) we repeat this operations for threads matching the criterion threadIdx.x%2^i==0 and sum the result from threadIdx.x+2^(i-1), until we have completed log2(N) operations. This works best for the block sizes which are power of 2, but this is typically our choice anyways.

Once the threads complete their local reductions, it is time to compute a global sum amongst the blocks. Again, we could just use atomicAdd, but the binary-addition algorithm outlined above could be applied to blocks as well, with the exception that we cannot use shared memory anymore. For the purpose of this exercise you can choose the number of blocks is a power of 2.

Implement this reduction kernel (which can actually be two kernels, if you prefer, one for inter-block, one for between-block reduction) and time the execution of the entire program for the range of values N the same as in Task 2. Compare the timings to precious results and write your conclusions.