Cholesky Decomposition on the GPU

Joel Vasanth

August 3rd, 2022

This short report presents the speed-up achieved by GPU acceleration of a sample linear algebra algorithm - the Cholesky Decomposition. It was submitted to the OpenACC and Hackathons Summit 2022 (link), held on August 2nd-4th 2022. The code is written in C and is in the file `cholesky_parallel.c` with documentation for each function. Kindly read this report in conjunction with `cholesky_parallel.c`.

1 What is the Cholesky Decomposition (CD)

The CD is a linear algebra operation to convert a real symmetric matrix or a Hermitian matrix (if its entries are complex) A of size N into the product of two matrices - a lower triangular matrix L and its conjugate transpose L^*. In equation format: $A = LL^*$.

The formula followed for the CD is

$$\begin{aligned} L_{ij} &= \sqrt{A_{ii} - \sum_{k=0}^{i-1} L_{ik}^2} \quad \text{if} \quad i = j \\ L_{ij} &= \frac{A_{ij} - \sum_{k=0}^{i-1} L_{ik} L_{kj}}{L_{jj}} \quad \text{if} \quad i > j, \end{aligned} \tag{1}$$

where i and j are row and column indices respectively.

2 Initialisation and Parallelisation

I have written a serial algorithm and an OpenACC parallelised version for the CD as described above. The matrix sizes I have tried are $N = 10,\ 100,\ 1000$.

```
1.0000 0.0100 0.0200 0.0300 0.0400 0.0500 0.0600 0.0700 0.0800 0.0900
0.0100 1.0000 0.0300 0.0400 0.0500 0.0600 0.0700 0.0800 0.0900 0.1000
0.0200 0.0300 1.0000 0.0500 0.0600 0.0700 0.0800 0.0900 0.1000 0.1100
0.0300 0.0400 0.0500 1.0000 0.0700 0.0800 0.0900 0.1000 0.1100 0.1200
0.0400 0.0500 0.0600 0.0700 1.0000 0.0900 0.1000 0.1100 0.1200 0.1300
0.0500 0.0600 0.0700 0.0800 0.0900 1.0000 0.1100 0.1200 0.1300 0.1400
0.0600 0.0700 0.0800 0.0900 0.1000 0.1100 1.0000 0.1300 0.1400 0.1500
0.0700 0.0800 0.0900 0.1000 0.1100 0.1200 0.1300 1.0000 0.1500 0.1600
0.0800 0.0900 0.1000 0.1100 0.1200 0.1300 0.1400 0.1500 1.0000 0.1700
0.0900 0.1000 0.1100 0.1200 0.1300 0.1400 0.1500 0.1600 0.1700 1.0000
```

Figure 1: A sample real-symmetric matrix for $N = 10$ as a result of `initmult()`. This is one of the input matrices on which CD is applied.

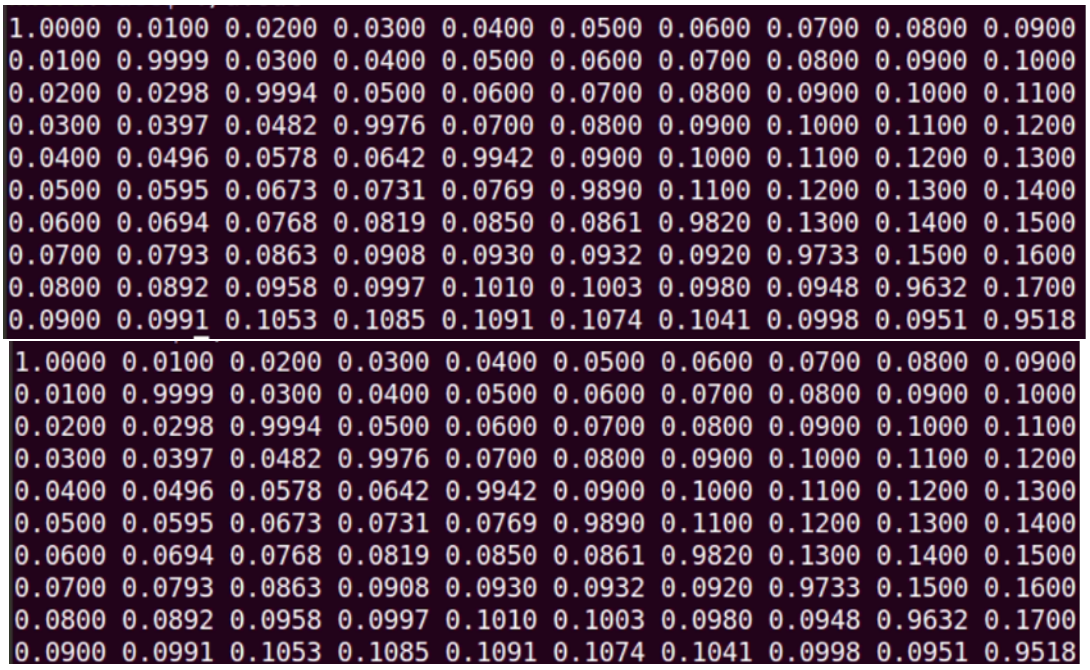

Figure 2: Matrix A after CD is applied in serial (above) and parallel (below) for N = 10.

The code is in `cholesky_parallel.c`, with comments for each function. In Fig. 1 is the initialized matrix for $N = 10$, as run by the serial code. This is the result of the `initmult()` function.

I first parallelize `initmult()`. I use the `parallel loop` directive along with the present clause to prevent unnecessary copying in of A. The matrix is created on the GPU itself. This is made possible by the create clause in the `initmult()` function. Both loops in `main()` are successfully parallelized since the initialization by each threads has no loop carried or data dependencies.

The function `cholesky()` is parallelized next. In this function, there are three nested loops for the off diagonal elements. There are data dependencies for the first two outer loops, but not for the third. The innermost loop sums up already computed values in the ith row and the jth row to determine new elements of the decomposition in the ith row. Thus, a parallel loop directive along with a reduction clause can be used. A similar directive can be applied for the loop over the diagonal elements. I use a `parallel` directive at the beginning of the outermost loop. The compiler automatically correctly detects that the innermost loop requires parallelization with `reduction`, whereas the two outer loops have data dependencies and thus are run in serial. The parallelisability of the loop over the diagonal elements is also automatically identified and that loop is parallelized.

3 Results

By performing the above parallelization, the same sample matrix from Fig. 1 is decomposed using this function and the output is printed in Fig. 2. Serial and parallel outputs are compared. Note that A is overwritten to contain the solution. The solution of the decomposition

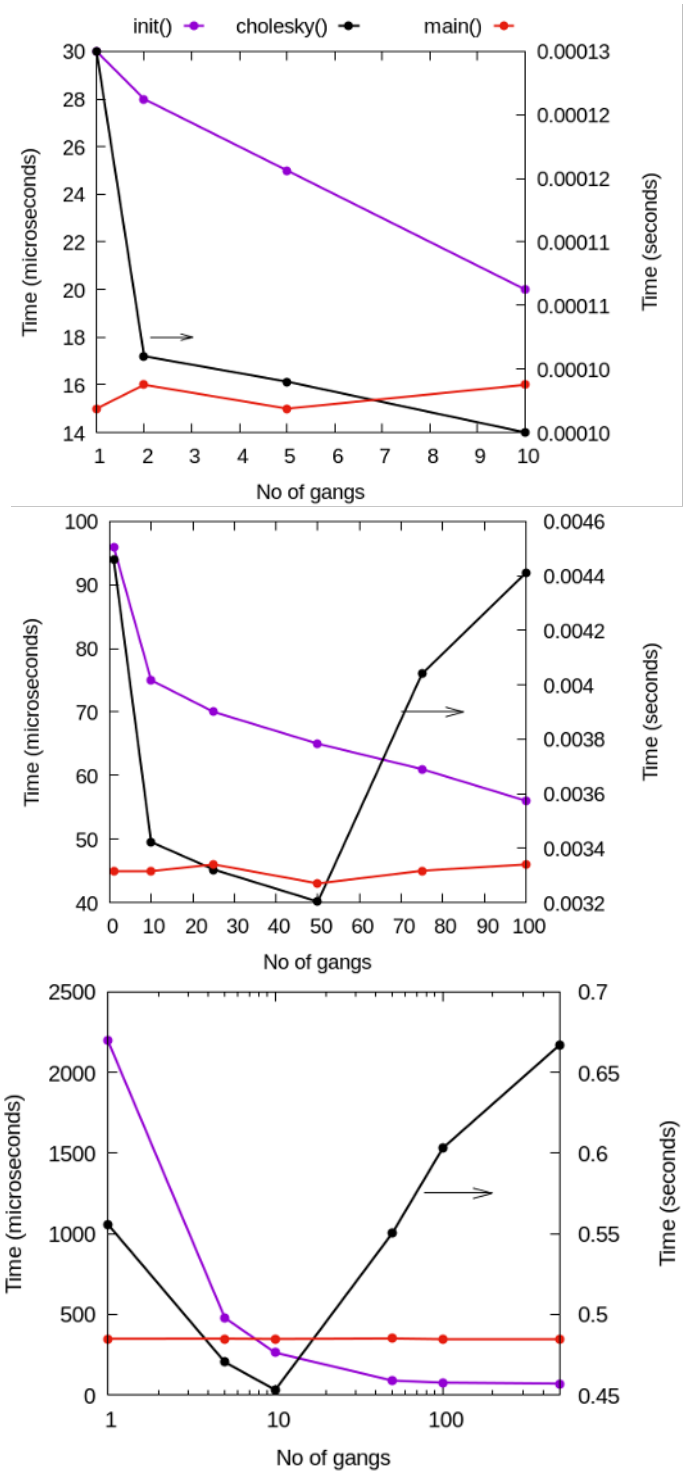

Figure 3: (Above to below: $N = 10,\ 100,\ 1000$) Plots of runtimes for different functions of `cholesky_parallel.c` vs the number of gangs N_G (horizontal axis). The timing for the `cholesky()` function alone (black) is plotted on the right-vertical axis. The `initmult()` function is renamed as `init()` here for brevity.

is printed in the lower triangular part of the Fig. 2, i.e. $j < i$. The solution matches that computed by the serial case exactly, showing that the parallel implementation is correct.

The runtimes are tested by changing the number of gangs in `cholesky_parallel.c`. For the serial version, the time taken for $N = 10$ is 155 microseconds, for $N = 100$ is 2.1 ms for $N = 1000$ is 1.03 s. For the parallel code, I plot the different times taken by the different subroutines

and as a function of the number of gangs in Fig 3.

I first analyse the $N = 10$ case (top most panel). The `initmult()` function time reduces as N_G reduces, since the routine consists of two nested loops, without any loop carried dependencies that parallelize well with the number of gangs. The `cholesky()` function is also seen to reduce in time as the number of gangs increases. Thus, an optimum for the $N = 10$ case seems to be 10 gangs. The `main()` function does not change time, since there is no block of code in `main()` that is parallelized. Every part of `main()` runs on the host (except for the functions it calls). This observation of the `main()` function holds good for $N = 100$ and 1000 cases as well.

For the $N = 100$ and 1000 cases, the same trend is observed for `initmult()` and `main()` for the same reasons cited above. The times are much larger than in the $N = 10$ case, due to the size of the problem. The trend of the `cholesky()` function, however, shows some difference from the $N = 10$ case. For a number of gangs higher than 50, there is an increase in the run time rather than a decrease. This is because of the nature of the parallelization. The parallel loop is the innermost nested loop in a set of three nested loops.

A synchronization barrier is implied at the end of the `parallel` loop directive. Since this occurs in the innermost nested loop, the barrier is encountered $N^2/2$ times. After some gangs have finished computation, they have to wait for the unfinished gangs to resume the loop. When the number of gangs are low, the trade-off between the wait times and the work-sharing among the gangs is such that a speed-up is achieved. When the gangs are higher than a critical value (50, for $N = 100$ and 10 for $N = 1000$), the speedup is not as much as when the gangs are lower than this value.