Hybrid MPI CUDA N-Qubit Simulation

Christopher M. Cantwell Jose Gonzalez

November 25, 2013

1 Introduction

2 Quantum State Evolution

3 Parallelization

3.1 Step Matrix

Given the unitary evolution operator $U(t) = e^{-iHt}$ a quantum state evolves as $|\psi(t+\Delta t)\rangle = U(\Delta t)\,|\psi(t)\rangle$. We use a time symmetric representation and expand the exponential to first order in order to solve for a step matrix S.

$$(I + iH\frac{\Delta t}{2})\,|\psi(t+\Delta t)\rangle = (I - iH\frac{\Delta t}{2})\,|\psi(t)\rangle \tag{1a}$$

$$|\psi(t+\Delta t)\rangle = (I + iH\frac{\Delta t}{2})^{-1}(I - iH\frac{\Delta t}{2})\,|\psi(t)\rangle \tag{1b}$$

Thus we see that $S = (I - iH\frac{\Delta t}{2})^{-1}(I + iH\frac{\Delta t}{2})$ which can be solved by using LU decomposition.

$$(I - iH\frac{\Delta t}{2})S = (I + iH\frac{\Delta t}{2}) \tag{2a}$$

$$(I - iH\frac{\Delta t}{2}) = LU \quad , \quad (I + iH\frac{\Delta t}{2}) = B \tag{2b}$$

$$LUS = B \tag{2c}$$

$$LY = B \tag{2d}$$

$$US = Y \tag{2e}$$

So we need to parallellize three tasks. First decompose $(I - iH\frac{\Delta t}{2})$ into LU. Second solve $LY = B$ for Y. And finally solve $US = Y$ for S. So long as the Hamiltonian is time independent this only needs to be done once at he start of the program.

3.2 State Evolution

Consider the equation $Ax = b$ where A is a matrix and x and b are vectors. We can divide the matrix and vectors into blocks as follows:

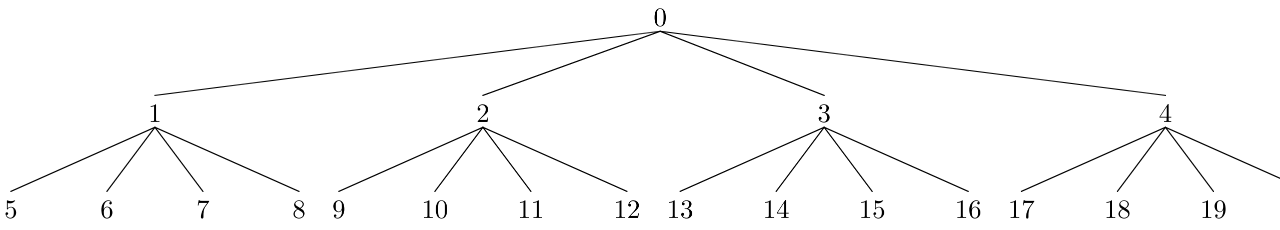
$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \left(\begin{array}{c} x_1 \\ \hline x_2 \end{array}\right) = \left(\begin{array}{c} b_1 \\ \hline b_2 \end{array}\right) \tag{3}$$

We can see clearly that

$$b_1 = A_{11}x_1 + A_{12}x_2 \tag{4a}$$

$$b_2 = A_{21}x_1 + A_{22}x_2 \tag{4b}$$

This can be parallellized by dividing it into four process, each calculating one of the matrix-vector products from the above equations. Subdividing again leads to a natural tree structure. For an N qubit system vector x has 2^N elements.



At each level we divide the vector into two equal size blocks. Thus at level l the vector has 2^{N-l} elements. For efficiency we want this vector and the matrix it is being multiplied by to fit in the memory available on the GPU. The Nvidia Kepler K20 has 5G of on chip memory. Assume a dense matrix and vector with complex double precision elements. We need 16 bytes to represent a single element. To store A, x, and b at level l we need $2^{2(N-l)} + 2^{N-l+1}$ elements which gives us

$$(2^{2(N-L)} + 2^{N-L+1}) elements \times 8 \frac{bytes}{element} = 2^{2(N-L)+3} + 2^{N-L+4} bytes$$

To determine the number of levels we must go to for the data to fit in GPU memory we have:

$$\begin{aligned}
2^{2(N-L)+3} + 2^{N-L+4} &\leq 5 \times 10^6 \\
Log_2(2^{2(N-L)+3} + 2^{N-L+4}) &\leq Log_2(5 \times 10^6) \\
Log_2(2^{N-L+4}(2^{2(N-L)+3-(N-L+4)} + 1)) &\leq \frac{Log_{10}(5 \times 10^6)}{Log_{10}(2)} \\
2^{N-L-1} + 1 < 2^{N-L} \quad &and \quad \frac{1}{Log_{10}(2)} < \frac{1}{0.3} \\
Log_2(2^{N-L+4}) + Log_2(2^{N-L}) &\leq \frac{1}{0.3} \times (Log_{10}(5) + Log_{10}(10^6)) \\
N - L + 4 + N - L &\leq \frac{1}{0.3} \times (Log_{10}(5) + Log_{10}(10^6)) \\
2N - 2L + 4 &\leq \frac{1}{0.3} \times (0.7 + 6) \\
2N - 2L + 4 &\leq 22.3 \\
2L &\geq 2N + 4 - 22.3 \\
L &\geq N - 9.15 \\
L &\geq N - 9
\end{aligned}$$

This gives us a total number of processes of

$$T_P = \sum_{l=0}^{N-7} 4^l \tag{5}$$

4 The Algorithm

```
calculate $L$, total number of levels in the tree, based on GPU memory
size;
calculate $M$, max level for parallel processing, based on number of nodes;
initialize id;
iniitialize nsteps;
$step \leftarrow 0$;
$U[0 : 4^{L-M}] \leftarrow$ submatrix indices to be used for state evolution;
if $(id == 0)$ then
    $initialize(state)$;
end
while $step \neq nsteps$ do
    $l \leftarrow 0$;
    parition(state);
    $s[0 : 4^{L-M}] \leftarrow partition(state, 4^{L-M})$;
    $n \leftarrow 0$;
    while $(n \neq 4^{L-M})$ do
        $s[n] \leftarrow gpu\_mv\_mult(U[n], s[n])$;
        $n + +$;
    end
    while $(n \neq 4^{M})$ do
        $state \leftarrow recombine(s[n], s[n-1], s[n-2], s[n-3])$;
        $n \leftarrow n - 4$;
    end
    while $l \neq 0$ do
        if $master$ then
            $s[1] \leftarrow receive(children[1])$;
            $s[2] \leftarrow receive(children[2])$;
            $s[3] \leftarrow receive(children[3])$;
            $state \leftarrow recombine(s[0], s[1], s[2], s[3])$;
            $send(state, parent)$;
        else
            $send(state, master)$;
        end
        $l - -$;
    end
    $step + +$;
end
```

```
$partition(state)$ while ($l \neq M$) do
    if ($master$) then
        $children[0:2] \leftarrow$ MPI communicator subgroup;
        $p[0:3] \leftarrow partition(state, 4)$;
        $send(p, children)$;
    else
        $state \leftarrow receive(parent)$;
    end
    $l++$;
end
```

5 Complexity Analysis

To analyze complexity we assume a sequential algorithm using a single Kepler K20 GPU and a system size of N qubits.

$$T_{seq} \propto 2^{N-10} \times (T_{gpu\_mult} + T_{gpu\_comm}) \tag{6}$$

For our parallel MCQE algorithm we have the following.

$$T_{MCQE} \propto 2l \times T_{mpi\_comm} + \frac{2^{N-l-9}}{4^l} \times (T_{gpu\_mult} + T_{gpu\_comm}) \tag{7}$$

where 4^l is the number of MPI ranks available to us. In general T_{mpi_comm} will depend on the current tree level as at lower levels there will be less data to transfer. This leads to a speedup of

$$\frac{l \times T_{mpi\_comm} + 2^{N-2l-10} \times (T_{gpu\_mult} + T_{gpu\_comm})}{2^{N-10} \times (T_{gpu\_mult} + T_{gpu\_comm})} \tag{8}$$

6 Results

7 Scratch

To reduce the overhead caused by MPI communication we can take advantage of the fact that the GPU and CPU can run independently. We assume that δt is small enough that the subvector being used at a given node k does not change much over some number of time steps N_{steps}. While the CPU is handling MPI communication and recombining results the GPU can continue state evolution. After N_{steps} the recombined and subsequently partitioned state vector is returned to the node. We can find the error between the vector used at node k and what the vector should have been. We can then apply an error correcting procedure and repeat the process.

$$\begin{aligned} S' &= U^{N_{steps}}(S + \epsilon) && \text{(9a)} \\ &= U^{N_{steps}} S + U^{N_{steps}} \epsilon && \text{(9b)} \\ & && \text{(9c)} \end{aligned}$$

From the above we can see that if, after node k receives the results of recombination and repartitioning, we calculate ϵ and apply a correction we can retreive the correct result. While applying U N_{steps} times would retreive the exact result this would not provide any speedup. But because ϵ is small we can probably get away with less.