Quantum Computer Simulation Using CUDA (Quantum Fourier Transform Algorithm)

Alexander Smith & Khashayar Khavari

Department of Electrical and Computer Engineering
University of Toronto

April 15, 2009

Outline

Brief Overview and Motivation

- Quantum computers achieve dramatic speedups by exploiting superposition at quantum level
- Slow advances in building a quantum computer hampers algorithm development
- Simulation of quantum algorithms on classical machines are extremely time consuming

- *linquantum*
  - A set of libraries written in C for quantum computer simulation
  - Implementation of many well-known algorithms
  - Easy development of new quantum algorithms
  - Gate-by-gate simulation of quantum circuits

Brief Overview and Motivation

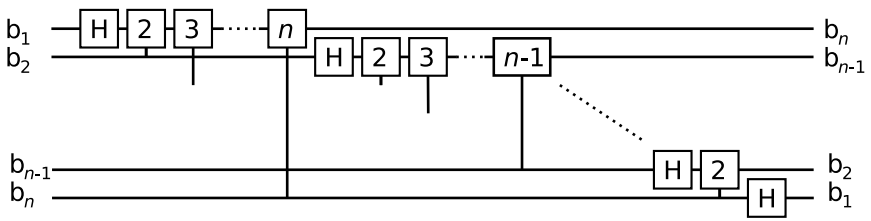
- Quantum computers achieve dramatic speedups by exploiting superposition at quantum level
- Slow advances in building a quantum computer hampers algorithm development
- Simulation of quantum algorithms on classical machines are extremely time consuming

- *linquantum*
  - A set of libraries written in C for quantum computer simulation
  - Implementation of many well-known algorithms
  - Easy development of new quantum algorithms
  - Gate-by-gate simulation of quantum circuits

Quantum Fourier Transform (QFT)

- Used by many quantum algorithms including Shor's factorization
- Made up of conditional phase shifts and Hadamard transforms
- Circuit consists of $\frac{n(n-1)}{2}$ gates
- Input and output are weighted sums of 2^n quantum states (vectors of 2^n complex numbers)

$$\Psi_{in} = \sum_{i=0}^{2^{n-1}} \alpha_i |i>, \qquad with: \quad |i> = [0 \ldots 0\ 1\ 0 \ldots 0]^T$$



QFT as Matrix Multiplication

- Any quantum circuit of n bits can be reduced to matrix-vector multiplication
- eg. $n = 3$, $w = e^{2\Pi i/8}$

$$\frac{1}{\sqrt{8}} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ 1 & w^2 & w^4 & w^6 & 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w & w^4 & w^7 & w^2 & w^5 \\ 1 & w^4 & 1 & w^4 & 1 & w^4 & 1 & w^4 \\ 1 & w^5 & w^2 & w^7 & w^4 & 1 & w^6 & w^3 \\ 1 & w^6 & w^4 & w^2 & 1 & w^6 & w^4 & w^2 \\ 1 & w^7 & w^6 & w^5 & w^4 & w^3 & w^2 & w \end{bmatrix}$$

- Memory access is coalesced when multiplying but not when adding
- Achieved 1000 times speedup compared to CPU version
- Complexity is $O(2^n \times 2^n)$
- Dramatically slower than the gate-by-gate simulation

Gate-by-Gate Simulation

- Complexity is $O(2^n \times n^2)$
- Coefficient of output state i corresponding to q^{th} bit is a function of input coefficients i and $i \oplus 2^q$

$$\Psi_{out} = \sum_{i=0}^{2^{n-1}} \beta_i |i>, \qquad where \quad \beta_i = f(\alpha_i, \alpha_{i \oplus 2^q})$$

- Hadamard transforms result in all possible combinations of phase shifts

- Simple implementation
  - n Kernel calls
  - One thread per state (2^n threads per kernel call)
  - Each thread finds output after application of multiple gates

Gate-by-Gate Simulation

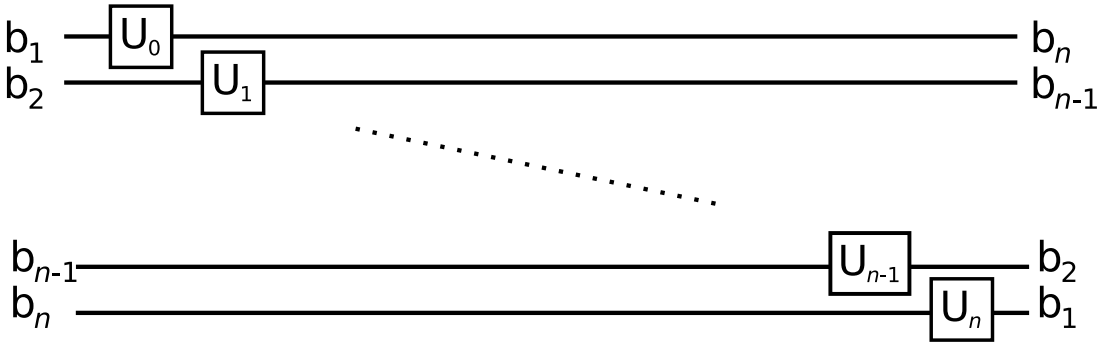
- Complexity is $O(2^n \times n^2)$
- Coefficient of output state i corresponding to q^{th} bit is a function of input coefficients i and $i \oplus 2^q$

$$\Psi_{out} = \sum_{i=0}^{2^{n-1}} \beta_i |i>, \qquad where \quad \beta_i = f(\alpha_i, \alpha_{i \oplus 2^q})$$

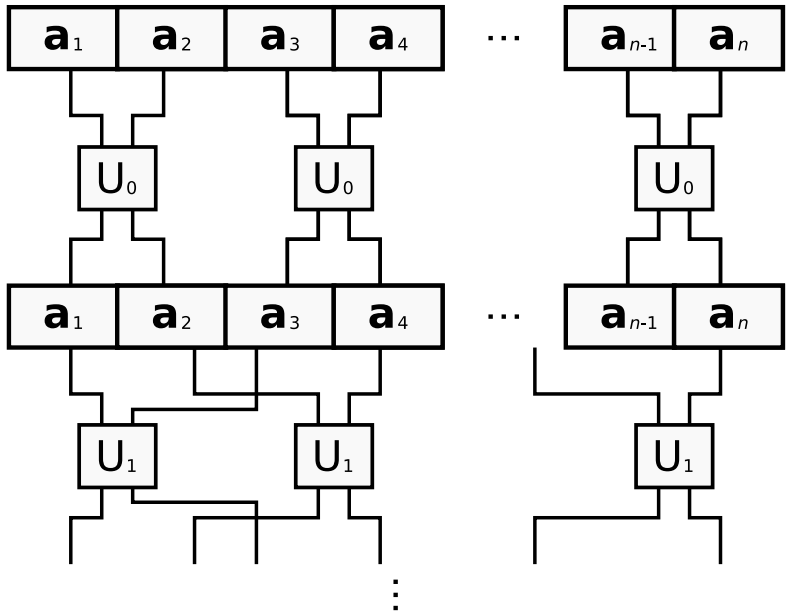
- Hadamard transforms result in all possible combinations of phase shifts

- Simple implementation
  - n Kernel calls
  - One thread per state (2^n threads per kernel call)
  - Each thread finds output after application of multiple gates

QFT Circuit and Memory Access



U_q reads and writes a_i and $a_{i \oplus 2^q}$ $(i = \sum_{p=1}^{n} b_p 2^p)$



Evaluation of the Simple Approach

- Performance - Initial implementation

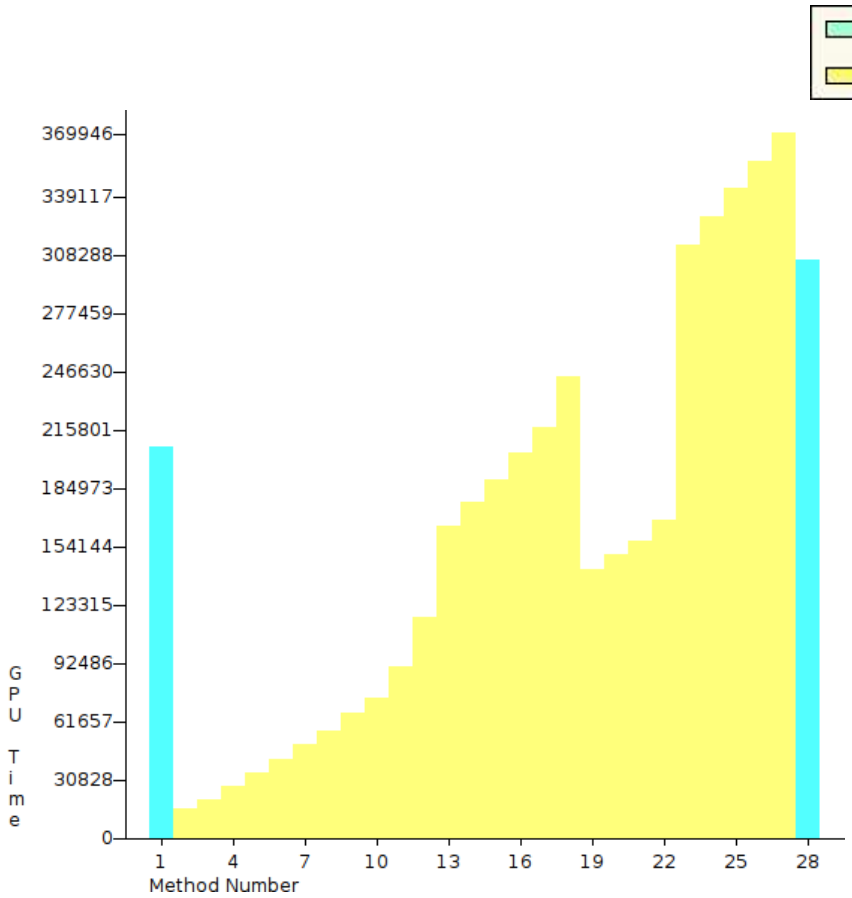
bits	10	12	14	16	18	20	22	24	26
CPU [ms]	0	1.915	10.51	40.42	247.0	1342	6272	29163	134627
GPU [ms]	0.23	0.23	0.66	2.00	8.65	40.18	188.16	876.06	4096.00
Speedup	0	6.4	15.9	20.2	28.6	33.4	33.3	33.3	32.9

- Optimization targets
  - Complex sin & cos computations when simulating phase shifts
  - Four memory reads and four writes per thread.
  - Memory access is scattered (array[i] & array[$i \oplus 2^q$])

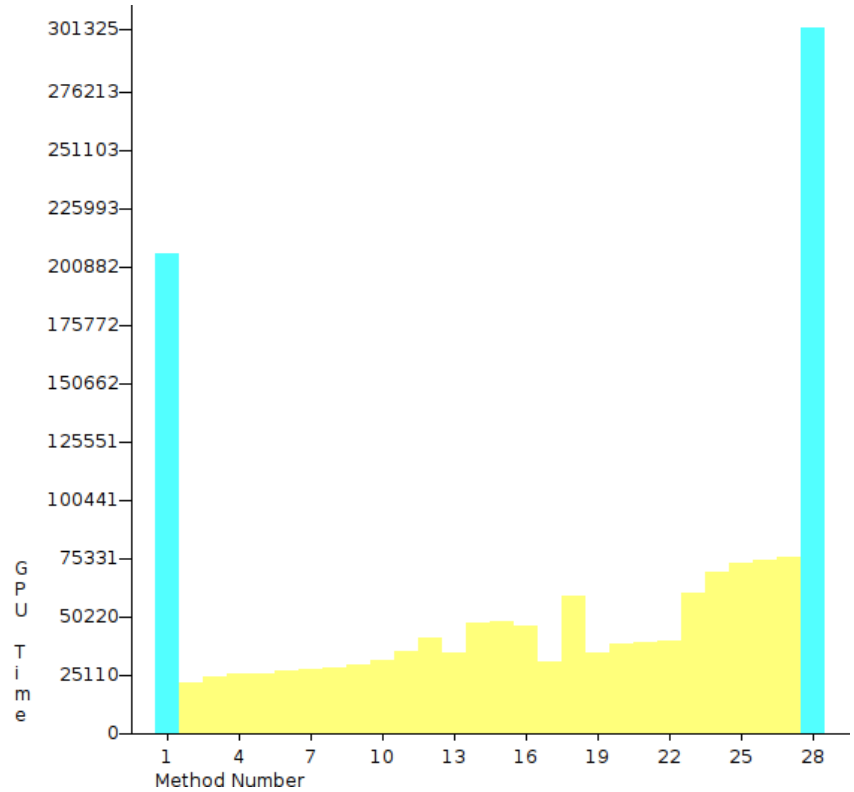
- Improvement strategy
  - Combine phase shift gates on a target bit into a single gate
  - Reduce global memory access
  - Combine multiple kernel calls

Combined Phase Shift

- A single evaluation of sin and cos per kernel



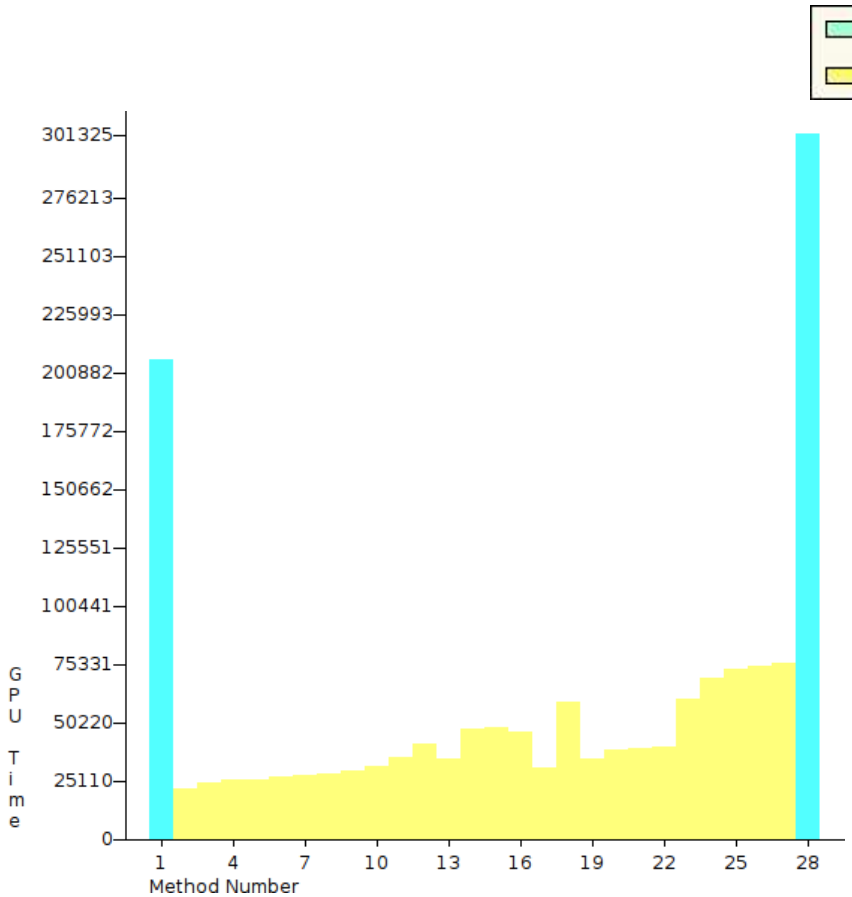
Simple gate-by-gate approach (v1).



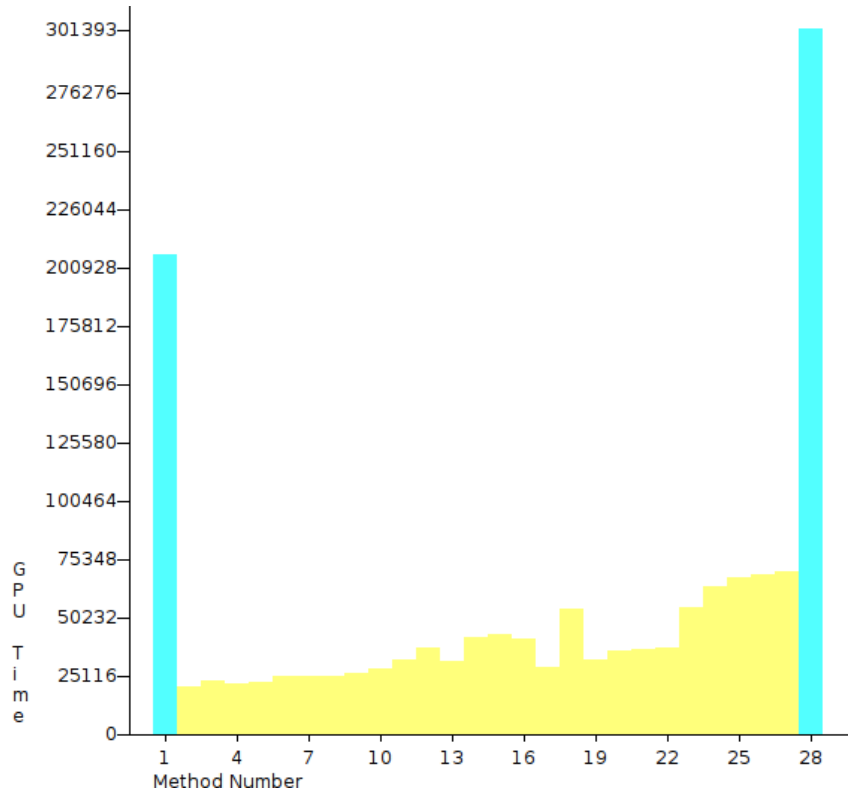
After combining phase shifts gates (v2).

Reducing Computation

- $sin(\phi) = \sqrt{1 - cos^2(\phi)}$



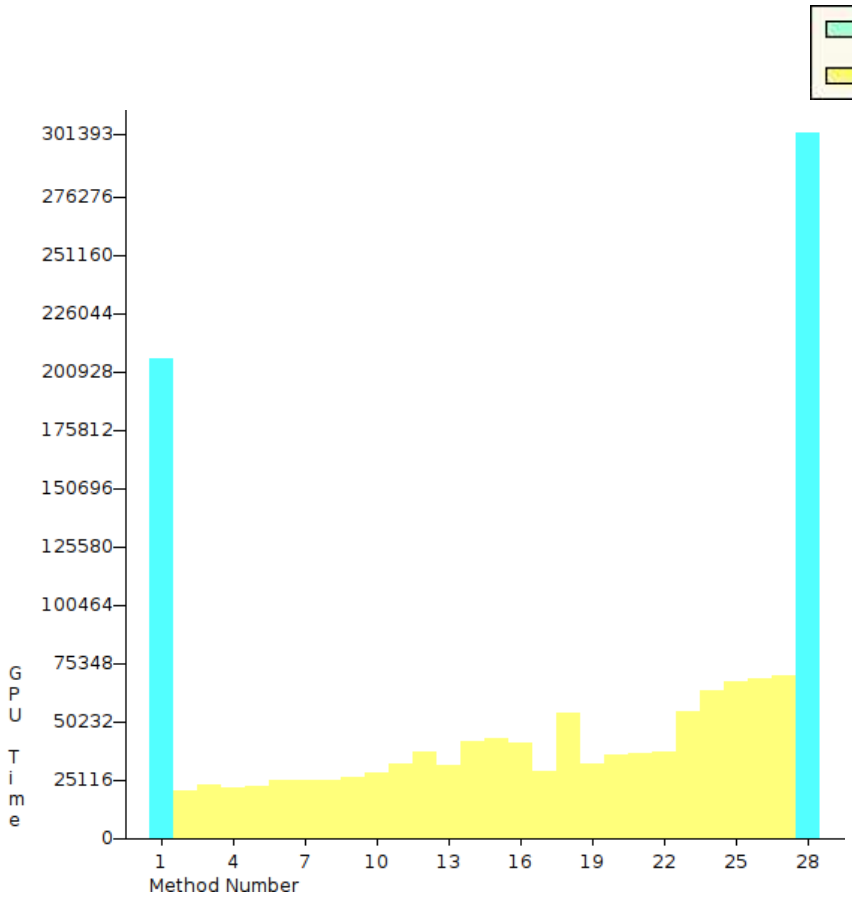
Combined phase shift gates (v2).



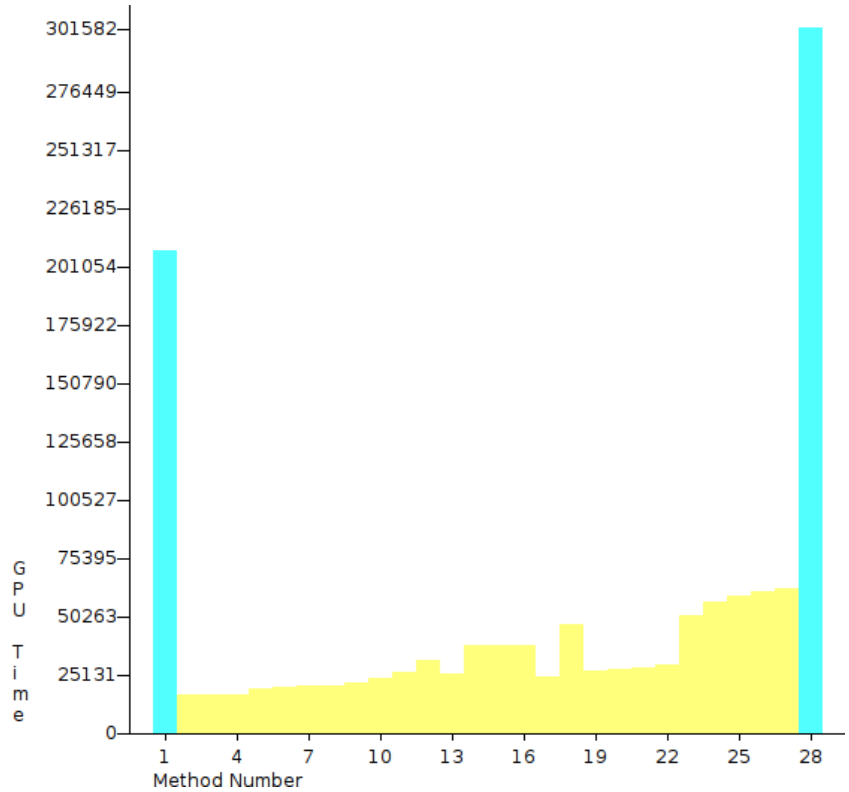
After eliminating one trigonometry computation (v3).

Reducing read and write operations

- Combining phase shift and Hadamard transform into one gate



Combined and optimized phase shift gates (v3).



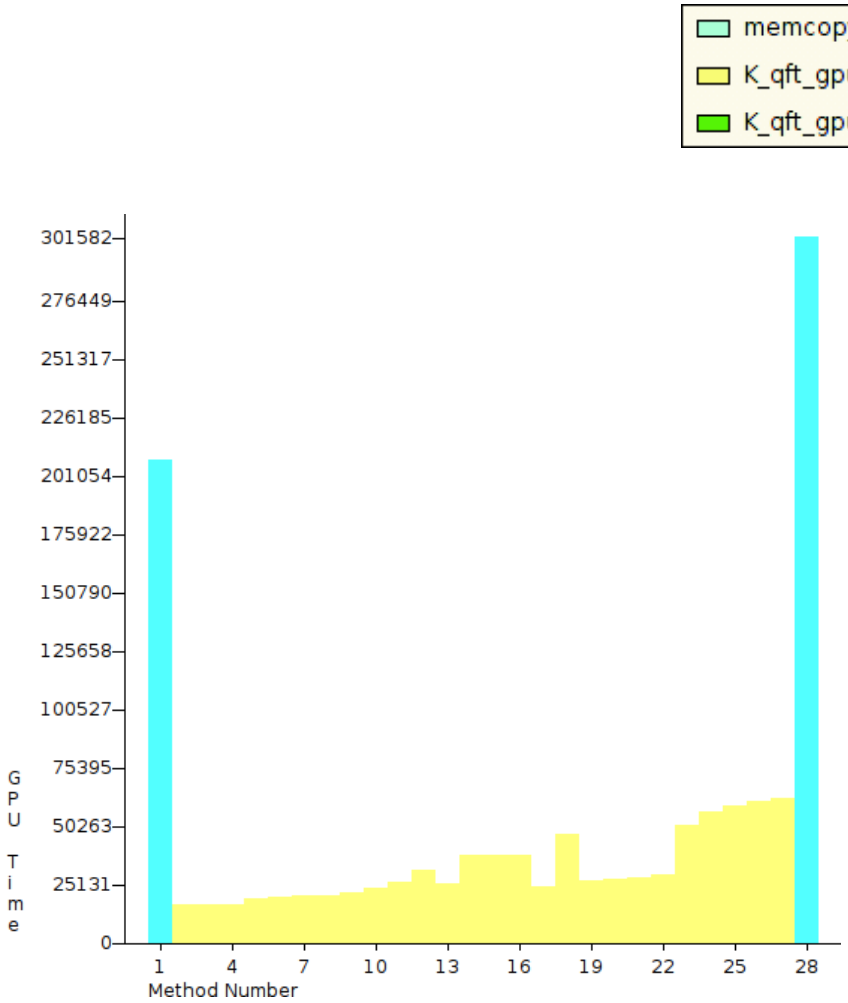
After combining all gates (v4).

Shared Memory and Reduction of Kernel Calls

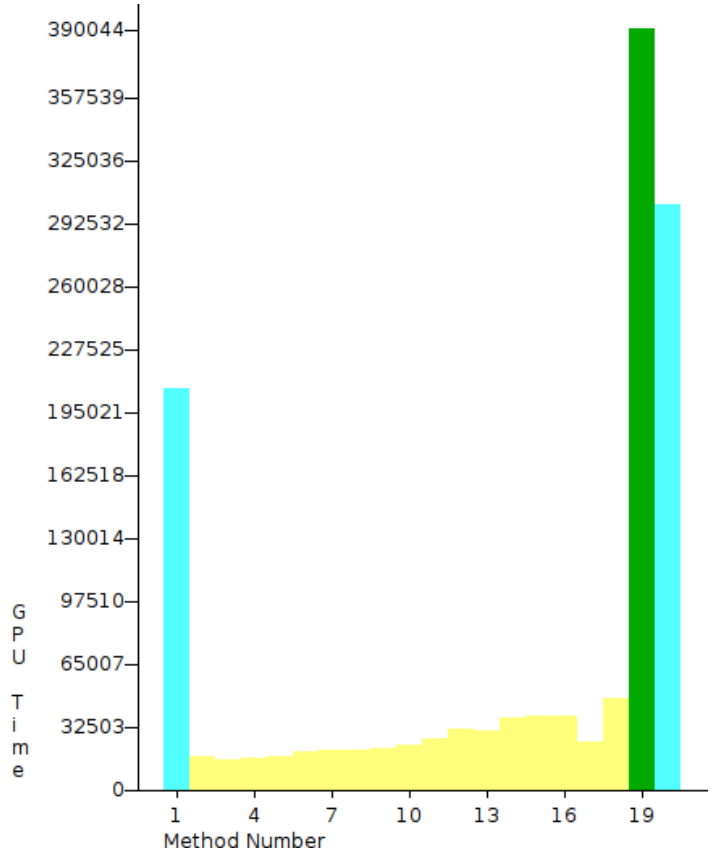
- The last 9 kernel calls use the same set of data.

q		0	1	...	7	8
	I	I xor 2^q	I xor 2^q	...	I xor 2^q	I xor 2^q
Block 1	0	1	2	...	128	256
	1	0	3	...	129	257
	2	3	0	...	130	258
	3	2	1	...	131	259
	4	5	6	...	132	260
	...	...	...	...	...	...
	511	510	509	...	383	255
Block 2	512	513	514	...	640	768
	513	512	515	...	641	769
	514	515	512	...	642	770
	515	514	513	...	643	771
	516	517	518	...	644	772
	...	...	...	...	...	...
	1023	1022	1021	...	895	767

Shared Memory - Last 9 Kernels



Gate-by-gate implementation with optimized phase (v4).



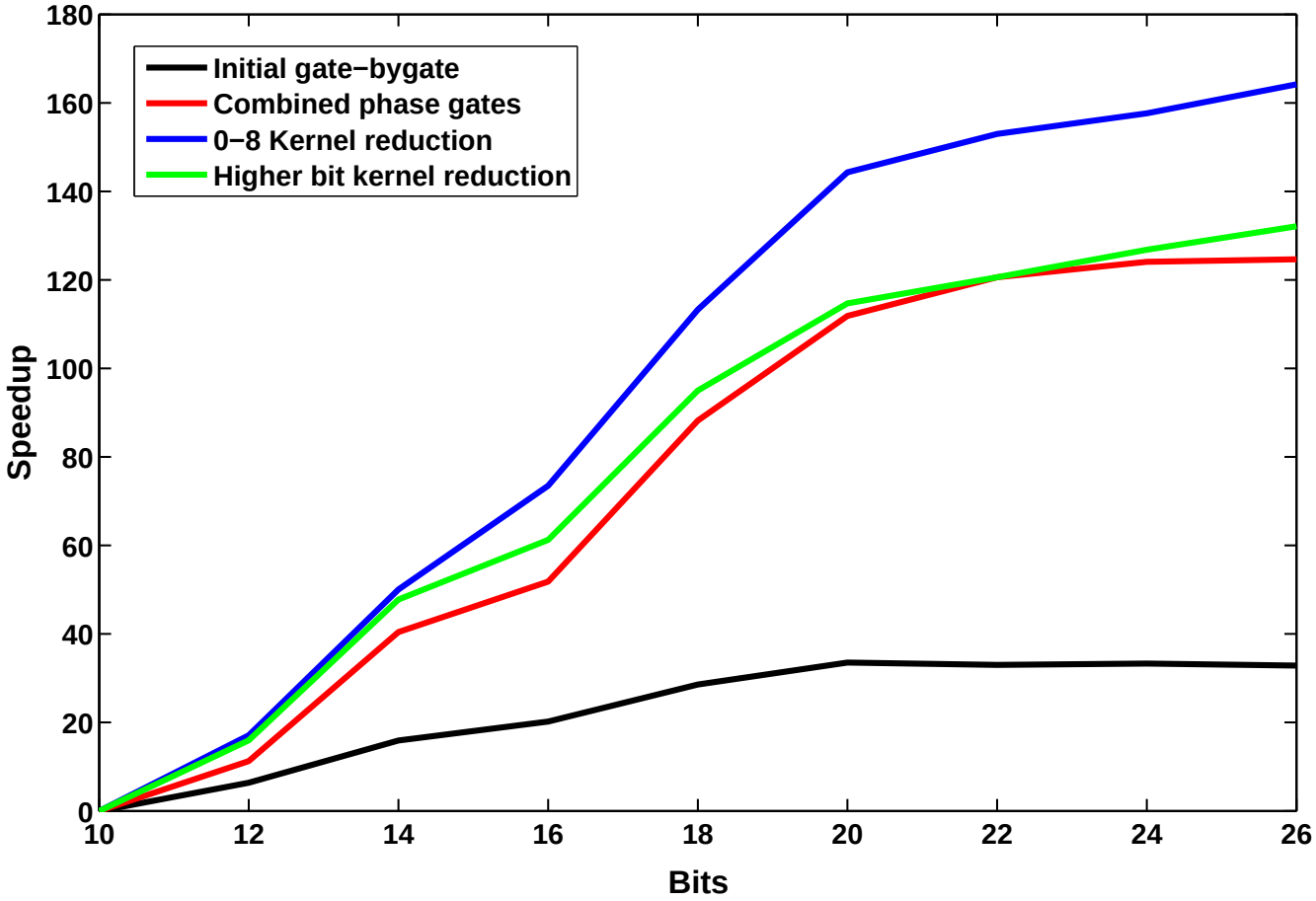
After replacing the last 9 kernel calls with one (v5).

Shared Memory - Higher Order Bits

- Closed set of memory accesses for constant number of serial kernel calls.

q		9	10	11
	I	I xor 2^q	I xor 2^q	I xor 2^q
Block 1	0	512	1024	2048
	...	...	...	...
	511	1023	1535	2559
Block 2	512	0	1536	2560
	...	...	...	...
	1023	511	2047	3071
Block 3	1024	1536	0	3072
	...	...	...	...
	1535	2047	511	3583
Block 4	1536	512	512	3584
	...	...	...	...
	2047	1023	1023	4096
Block 5	2048	2560	3072	0
	...	...	...	...
	2559	3071	3583	511
Block 6	2560	2048	3584	512
	...	...	...	...
	3071	2559	4096	1023
Block 7	3072	3584	2048	1024
	...	...	...	...
	3583	4096	2559	1535
Block 8	3584	3072	2560	1536
	...	...	...	...
	4096	3583	3071	2047

Evaluation


Initial gate−bygate
Combined phase gates
0−8 Kernel reduction
Higher bit kernel reduction
Speedup
Bits
0
20
40
60
80
100
120
140
160
180
10
12
14
16
18
20
22
24
26

Quantum Computer Simulation

- Optimization
  - Algebraic manipulation
  - Reduction of kernel calls
  - Use of shared memory
- Profiler results
  - Highly coalesced global memory access
  - Minimal branch divergence
  - Reduced warp serialization despite complex memory access patterns
- Speed up of 160X

Thank you!

Shor's Algorithm

- Integer factorization in polynomial time
- A probabilistic algorithm
- Conversion of factorization to period detection
  - $f_{a,N}(x) = a^x \% N\ , \qquad p = gcd(a^{r/2} - 1, N)$
- eg. $N = 15$, $a = 7$, from table $r = 4$ and $p = 3 \Rightarrow N = 3 \times 5$

x	0	1	2	3	4	5	6	7	...
f(x)	1	7	4	13	1	7	4	13	...

- Period detection is achieved through Quantum Fourier Transform

Shor's Algorithm

- Integer factorization in polynomial time
- A probabilistic algorithm
- Conversion of factorization to period detection
  - $f_{a,N}(x) = a^x \% N\ , \qquad p = gcd(a^{r/2} - 1, N)$
- eg. $N = 15$, $a = 7$, from table $r = 4$ and $p = 3 \Rightarrow N = 3 \times 5$

x	0	1	2	3	4	5	6	7	...
f(x)	1	7	4	13	1	7	4	13	...

- Period detection is achieved through Quantum Fourier Transform