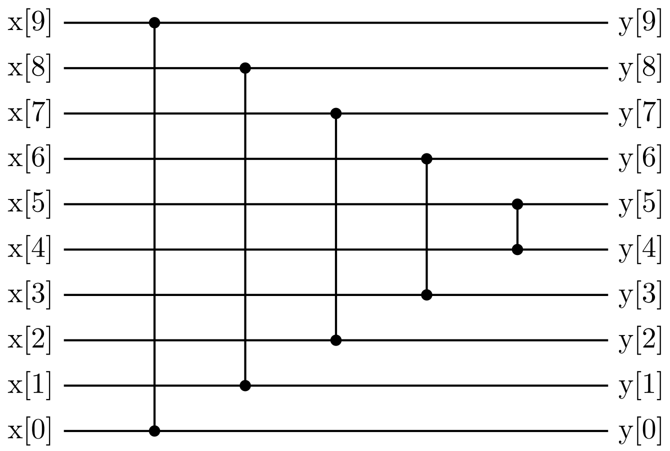
**CPSC 418: Parallel Computation** Apr 2, 2017

# Homework 4

## 1. Sorting Networks: clean and dirty sequences

(a) Draw this sorting network for the case N = 10.



(b) *Proof.* Define a sequence `u[0, ..., N - 1]` of 0-1 pairs based on the network and the inputs `x[0, ..., N - 1]`.

$$u[0,...,N-1] = \left\{ \begin{pmatrix} x[N-1] \\ x[0] \end{pmatrix}, \begin{pmatrix} x[N-2] \\ x[1] \end{pmatrix}, \cdots, \begin{pmatrix} x[N/2] \\ x[N/2-1] \end{pmatrix} \right\}.$$

Note that each pair inside `u[0, ..., N - 1]` corresponds to a compare-and-swap unit in the sorting network. They are ordered vertically and horizontally as in the diagram of (a).

Similarly, define an sequence `v[0, ..., N - 1]` based on the outputs `y[0, ..., N - 1]`.

$$\begin{aligned} v[0,...,N-1] &= \left\{ \begin{pmatrix} \max(x[0],x[N-1]) \\ \min(x[0],x[N-1]) \end{pmatrix}, \begin{pmatrix} \max(x[1],x[N-2]) \\ \min(x[1],x[N-2]) \end{pmatrix}, \cdots, \begin{pmatrix} \max(x[N/2-1],x[N/2]) \\ \min(x[N/2-1],x[N/2]) \end{pmatrix} \right\} \\ &= \left\{ \begin{pmatrix} y[N-1] \\ y[0] \end{pmatrix}, \begin{pmatrix} y[N-2] \\ y[1] \end{pmatrix}, \cdots, \begin{pmatrix} y[N/2] \\ y[N/2-1] \end{pmatrix} \right\}. \end{aligned}$$

In the following discussions, we call `x/y[0, ..., (N/2) - 1]` ”the lower sequence” and `x/y[N - 1, ..., N/2 - 1]` ”the upper sequence” (note the ordering is reversed for the upper sequence). According to the assumption, we have the input lower sequence in ascending order and the input upper sequence in descending order.

Given these two sequences, the input pair sequence `u[0, ..., N - 1]` can be divided into three parts: `u[0, ..., i]`, `u[i + 1, ..., j]` and `u[j + 1, ..., N-1]`.

The subsequence `u[0, ..., i]` has $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ pairs; `u[i + 1, ..., j]` has $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ or $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ pairs; `u[j + 1, ..., N-1` has $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ pairs . The index `i` and `j` indicates the position where the upper **or** the lower sequence change values.

Two example inputs,

$$\texttt{u[0, ..., N - 1]} = \left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \cdots, \begin{pmatrix}0\\1\end{pmatrix}\right\},$$
$$\texttt{u[0, ..., N - 1]} = \left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \cdots, \begin{pmatrix}0\\1\end{pmatrix}\right\}.$$

In both examples, `i = 1` and `j = 3`.

Depending on the inputs, it is possible to have one part out of three or two parts out of three to be empty. We will show the statement holds when all three parts are non-empty and generalize it to all possible cases.

When all three parts are non-empty, the first and the third parts only have $\begin{pmatrix}1\\0\end{pmatrix}$ pairs after sorting, and the middle part has two possible values depending on the inputs:

(i) The middle part has $\begin{pmatrix}0\\0\end{pmatrix}$ pairs as inputs. After sorting, the lower sequence has all 0s and is clean.

(ii) The middle part has $\begin{pmatrix}1\\1\end{pmatrix}$ pairs as inputs. The upper sequence has all 1s and is clean.

Thus, either the upper or the lower output sequence is clean when all three parts are non-empty.

When one part or two parts are empty, the input and output can be viewed as subsequences of the non-empty case. Any subsequence of a clean sequence is also clean. Therefore, either the upper or the lower output sequence is clean (or both when the second part is empty).

□

(c) *Proof.* As shown in (b), when all three parts are non-empty, one output would be clean and the other one be `{1, ..., 1, 0, ..., 0, 1, ..., 1}` (when the middle part has 0s) or `{0, ..., 0, 1, ..., 1, 0, ..., 0}` (when the middle part has 1s). These two output sequences are bitonic. When one part or two parts are empty, by the property of bitonic sequence: any subsequence of a bitonic sequence is bitonic, the outputs are still bitonic.

□

(d) *Proof.* In English, the question asks to prove that it is impossible to have 1 in the lower sequence when 0 is in the upper sequence.

Suppose there is at least a 1 in the lower output sequence and at least a 0 in the upper output sequence. Let the 1 be at position `i` and the 0 be at the position `j`.

Since these two numbers appear at their positions after sorting, there must be pairs $u[i] = v[i] = \begin{pmatrix}1\\1\end{pmatrix}$ and $u[j] = v[j] = \begin{pmatrix}0\\0\end{pmatrix}$ in the input. Otherwise, the 0 and the 1 would be swapped to the wrong sides.

Note that the assumption is that the input lower sequence is in ascending order and the input upper sequence is in descending order. Then, no matter $i < j$ or $i > j$, both upper and lower sequences have the $i$ and $j$ positions in the same ordering (1-0 or 0-1), which violates the assumption that two sequences are ordered differently (1-0 then the lower part is not ascending; 0-1 then the upper part is not descending). This is a contradiction. Therefore, every element of the lower half of `y` is less-than-or-equal-to any element of the top half.

□

## 1. Sorting Networks: clean and dirty sequences (Mark's version)

(a) See above.

(b) *Proof.* let $n_0 =$ number of zeros in `x[0, ..., N/2 - 1]` and $n_1 =$ number of zeros in `x[N/2, ..., N]`. Then, given `x[0, ..., N/2 - 1]` and `x[N/2, ..., N]` are in ascending order, `x[0, ...,` $n_0$ `- 1]` are zeros and `x[N/2, ..., N/2 +` $n_1$ `- 1]` are zeros.

(1) When $n_0 + n_1 \geq \frac{N}{2}$, that is, at least half inputs are zeros.
Consider $j$, such that $0 \leq j \leq \frac{N}{2} - 1$. In the output, $y[j] = \min(x[j], x[N-1-j])$.
If $j < n_0$, $x[j] = 0$ and $y[j] = \min(0, x[N-1-j]) = 0$.
If $j \geq n_0$, then $N - 1 - j > N - 1 - \frac{N}{2} = \frac{N}{2} - 1$ and

$$\begin{aligned} N - 1 - j &\leq N - 1 - n_0 \\ &\leq N - 1 - (\frac{N}{2} - n_1) \\ &= \frac{N}{2} + n_1 - 1 \\ &\leq \frac{N}{2} + n_1. \end{aligned}$$

Thus, $x[N-1-j] = 0$ and $y[j] = \min(x[j], 0) = 0$.
Therefore, the sequence `y[0, ..., N/2 - 1]` contains all zeros and is clean.

(2) When $n_0 + n_1 < \frac{N}{2}$, that is, there are more ones in the input.
Consider $j$, such that $0 \leq j \leq \frac{N}{2} - 1$. In the output, $y[N-1-j] = \max(x[j], x[N-1-j])$.
If $j > n_0$, $x[j] = 1$ and $y[N-1-j] = \max(1, x[N-1-j]) = 1$.
If $j \leq n_0$,

$$\begin{aligned} N - 1 - j &\geq N - 1 - n_0 \\ &> N - 1 - (\frac{N}{2} - n_1) \\ &= \frac{N}{2} - 1 + n_1. \end{aligned}$$

Thus, $x[N-1-j] = 1$ and $y[N-1-j] = \max(x[j], 1) = 1$.
Therefore, the sequence `y[N/2, ..., N - 1]` contains all ones and is clean.

As shown above, at least one of `y[0, ..., N/2 - 1]` and `y[N/2, ..., N - 1]` is clean.

□

(c) *Proof.* Consider cases similar to (b).

(1) When $n_0 + n_1 \geq \frac{N}{2}$. As shown in (b), the lower half contains all zeros and thus is bitonic.
Consider $j$, such that $0 \leq j \leq \frac{N}{2} - 1$. In the top half output, $y[N-1-j] = \max(x[j], x[N-1-j])$.
If $j < n_0$, $x[j] = 0$ and $y[N-1-j] = \max(0, x[N-1-j]) = x[N-1-j]$. There is no swapping in this part of top half sequence. Thus, `y[N -` $n_0$`, ..., N - 1]` is in ascending order.
If $j \geq n_0$, $y[N-1-j] = \max(1, x[N-1-j]) = 1$. `y[N/2, ..., N -` $n_0$ `- 1]` are all ones.
Combining `y[N/2, ..., N -` $n_0$ `- 1]` and `y[N -` $n_0$`, ..., N - 1]` generates a bitonic sequence.

(2) When $n_0 + n_1 < \frac{N}{2}$. As shown in (b), the top half contains all ones and thus is bitonic. Consider $j$, such that $0 \leq j \leq \frac{N}{2} - 1$. In the lower half output, $y[j] = \min(x[j], x[N-1-j])$.
If $j < n_1$, $x[N-1-j] = 1$ and $y[j] = \min(x[j], 1) = x[j]$ remains unswapped. Thus, `y[0, ..., `$n_1$` - 1]` is in ascending order.
If $j \geq n_1$, $x[N-1-j] = 0$ and $y[j] = \min(0, x[N-1-j]) = 0$. `y[N/2, ..., N - `$n_1$` - 1]` are all zeros.
Combining `y[N/2, ..., N - `$n_1$` - 1]` and `y[N - `$n_1$`, ..., N - 1]` generates a bitonic sequence.

As shown above, `y[0, ..., N/2 - 1]` and `y[N/2, ..., N - 1]` are bitonic.

□

(d) *Proof.* As shown in (b), depending on the inputs, there are two possible outputs: the sequence `y[0, ..., N/2 - 1]` containing all zeros and the sequence `y[N/2, ..., N - 1]` containing all ones. In either case, no `y` from the lower half is greater than any element of the top half.

□

## 2. Sorting Networks: examples

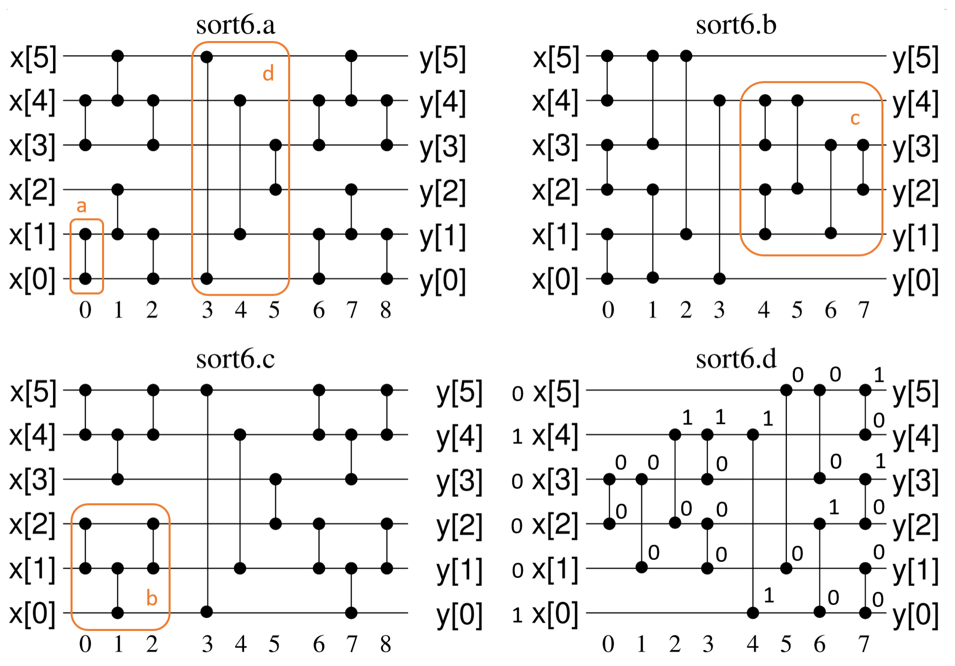
(a)
```
from sort6.a
   column 0:  (0,1)
```

(b)
```
from sort6.c
   column 0:  (1,2)
   column 1:  (0,1)
   column 2:  (1,2)
```

(c)
```
from sort6.b
   column 4:  (1,2), (3,4)
   column 5:  (2,4)
   column 6:  (1,3)
   column 7:  (2,3)
```

(d)
```
from sort6.a
   column 3:  (0,5)
   column 4:  (1,4)
   column 5:  (2,3)
```

(e) `sort6.d`, See image.



(f) `sort6.a` sorts correctly. We can show if at least one of `x[0,,5]` is a 1, then `y[5]` is a 1 since there is a path from `x[i]` to `y[5]` for `i` in `[0,,5]` formed by going up whenever possible and only traveling up and right. Here is a case to case analysis.

If `x[5]` is 1, then `y[5]` will be 1 regardless of the sorting network.

If `x[4]` is 1, col 1: (4,5) will make `y[5]` 1.

If `x[3]` is 1, col 0: (3,4), col 1: (4,5) will make `y[5]` 1.

If `x[2]` is 1, col 5: (2,3), col 6: (3,4), col 7: (4,5) will make `y[5]` 1.

If `x[1]` is 1, col 1: (1,2), col 5: (2,3), col 6: (3,4), col 7: (4,5) make `y[5]` 1.

If `x[0]` is 1, col 1: (1,2), col 5: (2,3), col 6: (3,4), col 7: (4,5) will make `y[5]` 1.

(g) Its fairly obvious that the 4 subnets at the four corners of sort6.a sorts 3 elements correctly. Thus, row 3 to 5 and row 0 to 2 are both sorted after column 2. By the results in question 1, the subnetwork in columns 3 to 5 will make sure any elements from row 3 to 5 is greater or equal to any element from row 0 to 2 afterwards as the inputs to the subnetwork are two sorted halves. Then, columns 6 to 8 make sure the row 3 to 5 are sorted and row 0 to 2 are sorted again. This makes the whole sorting network correct.

(h) Both `sort6.b` and `sort6.c` sort correctly. Explanation not needed here but is provided in brief. Its easily to see `sort6.c` is correct since we showed that `sort6.a` is correct and `sort6.c` is just a flip across the horizontal axis of `sort6.a`. For `sort6.b`, the first four columns ensures that the largest value get propagated to row 5, and the smallest value get propagated to row 0. Last four columns ensures that the four rows in the middle are sorted since the subnetwork sorts 4 elements correctly.

## 3. GPU GFlops

(a) We start off by choosing `n = 102,400` and `m = 100,000`. This is good enough that each trial of `f_test` takes less than 0.1s, but it's not so fast that our timing program counts the runtime as 0.0s. We're letting `n = 102,400` as it's a multiple of 8 (for the last method we try).

Here's the direct implementation:

```
for (int j = 0; j < m; j++) {
   x[i] = a*(x[i]*x[i]*x[i] - x[i]);
}
```

Without the -O3 flag over 50 trials, this gives 280-290 GFLOPS, and 300 GFLOPS with. (Using tpb=512) Note that GFLOPS are calculated as

$$\frac{\#\text{operations} \times n \times m}{\text{elapsed\_time} \times 10^9} = 4 \text{ nm/t}$$

We start off by choosing our tpb. Using the CUDA Occupancy Calculator (http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls) we find that `tpb = 512` gives 100% occupancy as a guess to start off with. This gives us a reasonably good result (as good as or better than other powers of 2 for tpb).

Next, we introduce loop unrolling by removing some overhead from the for loop above:

```
for (int j = 0; j < m/2; j++) {
   x[i] = a*(x[i]*x[i]*x[i] - x[i]);
   x[i] = a*(x[i]*x[i]*x[i] - x[i]);
}
```

Dividing the number of iterations by 2 gives 350 GFLOPS, by 4 gives a large jump to around 490 GFLOPS. After some testing, using 16 gives 530 GFLOPS; increasing from there doesn't seem to help.

Let's go back to the direct implementation with `tpb = 512`. We'll try something else now - unrolling the loop by allowing threads to access multiple adjacent elements. For example, let i be the unique id of a thread (equal to `blockIdx.x*blockDim.x + threadIdx.x`).

Then suppose each element accesses 2 adjacent elements, so we have:

```
for (int j = 0; j < m/2; j++) {
   x[8*i+0] = a*(x[8*i+0]*x[8*i+0]*x[8*i+0]-x[8*i+0]);
   x[8*i+1] = a*(x[8*i+1]*x[8*i+1]*x[8*i+1]-x[8*i+1]);
}
```

By doing this, notice that we increased the number of registers being used, from 7 in the previous method, to 17. (`nvcc --ptxas-options -v hw4.cu`) A kernel can use up to 21 registers per thread and still have 48 active warps. It seems like this method utilizes some caching mechanism for device global memory.

This gives 685 Gflops. These measurements were taken on `lin22`. It's specified that the GTX 550 Ti has peak Gflops of 691 (https://en.wikipedia.org/wiki/GeForce_500_series) but many machines in ICCS 005 are overclocked and the variance on speeds between machines can be high.

## 4. saxpy

(a) Measured on `lin07`. The value for `n` that I used for maximizing the throughput is 16777216 which took 3.288e-02 seconds, with a throughput of 5.103e+08 (16777216/3.288e-02). Some sample data is shown below. There are quite a bit of fluctuation even with 100 trials. In general the throughput is higher when n is larger.

(b) See implementation in `hw4.cu`. `t_elapsed` includes the three `memcpy` between CPU and GPU as well as the launched kernel. `saxpy_cpu(16777216, ...): t_elapsed = 1.244e-02, throughput = 1.349e+09` (average over 100 trials). The speedup I get is 1.244e-02/3.288e-02 = 0.378. The CPU version is faster than the GPU version.

(c) The limiting factors for the CUDA implementation of saxpy are the memory copying from the CPU to the GPU and GPU back to the CPU as well as memory accesses in the GPU. We can achieve 200Gflops easily in question 3. Using this conservative number for Gflops and ignoring memory accesses, we can process `saxpy` on 16777216 elements in 16,777,216*2/200,000,000,000 = 0.000167 seconds. However, the CUDA implementation took 3.288e-02 seconds to complete so it is clearly memory bound. See more data below (all averaged over 100 trials).

```
saxpy(256, ...):  t_elapsed = 1.200e-04, throughput = 2.133e+06.
saxpy(512, ...):  t_elapsed = 4.000e-05, throughput = 1.280e+07.
saxpy(1024, ...):  t_elapsed = 1.200e-04, throughput = 8.533e+06.
saxpy(2048, ...):  t_elapsed = 8.000e-05, throughput = 2.560e+07.
saxpy(4096, ...):  t_elapsed = 1.600e-04, throughput = 2.560e+07.
saxpy(8192, ...):  t_elapsed = 8.000e-05, throughput = 1.024e+08.
saxpy(16384, ...):  t_elapsed = 1.200e-04, throughput = 1.365e+08.
saxpy(32768, ...):  t_elapsed = 8.000e-05, throughput = 4.096e+08.
saxpy(65536, ...):  t_elapsed = 2.000e-04, throughput = 3.277e+08.
saxpy(131072, ...):  t_elapsed = 2.400e-04, throughput = 5.461e+08.
saxpy(262144, ...):  t_elapsed = 6.000e-04, throughput = 4.369e+08.
saxpy(524288, ...):  t_elapsed = 9.600e-04, throughput = 5.461e+08.
saxpy(1048576, ...):  t_elapsed = 2.280e-03, throughput = 4.599e+08.
saxpy(2097152, ...):  t_elapsed = 4.320e-03, throughput = 4.855e+08.
saxpy(4194304, ...):  t_elapsed = 8.760e-03, throughput = 4.788e+08.
saxpy(8388608, ...):  t_elapsed = 1.760e-02, throughput = 4.766e+08.
saxpy(16777216, ...):  t_elapsed = 3.288e-02, throughput = 5.103e+08.
saxpy(33554432, ...):  t_elapsed = 6.576e-02, throughput = 5.103e+08.
saxpy(67108864, ...):  t_elapsed = 1.308e-01, throughput = 5.129e+08.
saxpy_cpu(16777216, ...):  t_elapsed = 1.244e-02, throughput = 1.349e+09.
```