Lecture 3: CUDA Programming & Compiler Front End
Lecture 3A: GPU Programming
Review: Performance Hazard I — Control Divergence

• Thread Divergence Caused by Control Flows

How about this?

```
for (i=0;i<A[tid]; i++) {
  ...
}
```

OR this?

```
if (A[tid]) {
  do some work;
} else {
  do nothing;
}
```

May degrade performance by up to warp_size times.
(warp size = 32 in modern GPUs)

Optional Reading: Zhang et al. "On-the-Fly Elimination of Dynamic Irregularities for GPU Computing", ASPLOS’11
Review: Performance Hazard II — Global Memory Coalescing

• **Data is fetched as a contiguous memory chunk (a cache block/line)**
  - A cache block/line is typically 128 byte
  - The purpose is to utilize the wide DRAM burst for maximum memory level parallelism (MLP)

• **A thread warp is ready when all its threads’ operands are ready**
  - Coalescing data operands for a thread warp into as few contiguous memory chunks as possible
  - Assume we have a memory access pattern for a thread $... = A[P[tid]]$:

```plaintext
P[ ] = { 0, 1, 2, 3, 4, 5, 6, 7}
```
Review: Performance Hazard III — Shared Memory Bank Conflicts

• **Shared Memory**
  - Typically 16 or 32 banks
  - Successive 32-bit words are assigned to successive banks
  - A fixed stride access may cause bank conflicts
  - Maximizing bank level parallelism is important

• **Reduce Bank Conflicts**
  - Padding, might waste some shared memory space.
  - Data layout transformation, i.e., array of struct to structor of array.
  - Thread-data remapping, i.e., the tasks that are mapped to different banks are executed at the same time.
Review: Performance Hazard III — Shared Memory Bank Conflicts

- No Bank Conflicts

- No Bank Conflicts
Review: Performance Hazard III — Shared Memory Bank Conflicts

- 2-way Bank Conflicts

- 8-way Bank Conflicts
Case Study 2 — Parallel Reduction

• What is a reduction operation?
  - A binary operation and a roll-up operation
  - Let $A = [a_0, a_1, a_2, a_3, \ldots, a(n-1)]$, let $\oplus$ be an associative binary operator with identify element $I$

$$reduce(\oplus, A) = [a_0 \oplus a_1 \oplus a_2 \oplus a_3 \ldots a(n-1)]$$

```c
int sum = 0;
for ( int i = 0; i < N; i++ )
    sum = sum + A[i];
```

Sequential

Parallel
Case Study 2 — Parallel Reduction

• First Implementation

```c
int tid = threadIdx.x;
for ( s=1; s < blockDim.x; s *= 2) {
    if ( tid % (2*s) == 0 ) {
        sdata[ tid ] += sdata[ tid + s ];
    }
    __syncthreads();
}
```

Any GPU performance hazard here?

Thread Divergence
Case Study 2 — Parallel Reduction

• First Implementation

```c
tid = threadIdx.x;
for(s = 1; s < blockDim.x; s *= 2) {
  if(tid % (2*s) == 0) {
    sdata[tid] += sdata[tid+s];
  }
  __syncthreads();
}
```

Any GPU performance hazard here?

Thread Divergence

Reducing 16 Million Elements

**Implementation 1:**

Performance: Throughput = 1.5325 GB/s, Time = 0.04379 s
Case Study 2 — Parallel Reduction

• Reduce Thread Divergence

Thread-Task Mapping Changed!
Case Study 2 — Parallel Reduction

• Reduce Thread Divergence

```c
	tid = threadIdx.x;

	for (s=1; s < blockDim.x; s *= 2) {
		int index = 2 * s * tid;
		if (index < blockDim.x) {
			sdata[index] += sdata[index + s];
		}
	}
	syncThreads();
```

Second Version

```c

tid = threadIdx.x;

for (s=1; s < blockDim.x; s *= 2) {
	int index = 2 * s * tid;
	if (index < blockDim.x) {
		int sdata[index] += sdata[index + s];
	}
	syncThreads();
}
```

First Version

Thread-Task Mapping Changed!
Case Study 2 — Parallel Reduction

• Implementation 2

```c
int tid = threadIdx.x;
for (s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
}
__syncthreads();
```

Reducing 16 Million Elements

Implementation 1:
**Performance:** Throughput = 1.5325 GB/s, Time = 0.04379 s

Implementation 2:
**Performance:** Throughput = 1.9286 GB/s, Time = 0.03480 s

Potential Shared Memory Bank Conflicts
Case Study 2 — Parallel Reduction

• Reduce Shared Memory Bank Conflicts

Let Logically Contiguous Thread Access Contiguous Shared Memory Locations
Case Study 2 — Parallel Reduction

• Third Implementation (utilize implicit intra-warp synchronization)

```
tid = threadIdx.x;
for (s = blockDim.x / 2; s > 0; s >>= 1)
    if (tid < s)
        sdata[tid] += sdata[tid + s];
__syncthreads();
```

Reducing 16 Million Elements

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Throughput = 1.5325 GB/s, Time = 0.04379 s</td>
</tr>
<tr>
<td>2</td>
<td>Throughput = 1.9286 GB/s, Time = 0.03480 s</td>
</tr>
<tr>
<td>3</td>
<td>Throughput = 2.6062 GB/s, Time = 0.02575 s</td>
</tr>
</tbody>
</table>

Code: /ilab/users/zz124/cs515_2017/samples/6_Advanced/reduction
Review: Synchronization Primitive

• Within a thread block
  - Barrier: __syncthreads()
  - All computation by threads in the thread block before the barrier complete before any computation by threads after the barrier begins
  - Barriers are a conservative way to express dependencies
  - Barriers divide computation into phases
  - In conditional code, the condition must be uniform across the block

• Across thread blocks
  - Implicit synchronization at the end of a GPU kernel
  - A barrier in the middle of a GPU kernel can be implemented using atomic compare-and-swap (CAS) and memory fence operations

Optional Reading: Xiao et al. “Inter-block GPU communication via fast barrier synchronization”, IPDPS’10
Scan

- Let $A = [a_0, a_1, a_2, a_3, \ldots, a_{n-1}]$
- Let $\oplus$ be an associative binary operator with identity element $I$

\[
\text{scan}_{\text{inclusive}}(\oplus, A) = [a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots]
\]
\[
\text{scan}_{\text{exclusive}}(\oplus, A) = [I, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots]
\]

A prefix sum is $\text{scan}_{\text{inclusive}} \text{ scan}_{\text{inclusive}}(\oplus, A)$
Prefix Sum

- Let $A = [a_0, a_1, a_2, a_3, \ldots, a_{n-1}]$
- Let the binary operator $\oplus$ be $+$, $\text{prefix\_sum}(\oplus, A) = [a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots]$

Sequential Code:

```c
int PrefixSum[N];
PrefixSum[0] = 0;
for (i = 1; i < N; i++)
    PrefixSum[i] = PrefixSum[i-1] + A[i];
```

<table>
<thead>
<tr>
<th>$A[ ]$</th>
<th>3</th>
<th>1</th>
<th>7</th>
<th>0</th>
<th>4</th>
<th>1</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{PrefixSum}(+, A)$</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>15</td>
<td>16</td>
<td>22</td>
<td>25</td>
</tr>
</tbody>
</table>
How Would You Parallelize Scan?

• Recall the parallel reduction algorithm

A[ ]

\[
\begin{align*}
    a_0 &\quad a_1 &\quad a_2 &\quad a_3 &\quad a_4 &\quad a_5 &\quad a_6 &\quad a_7 &\quad a_8 &\quad a_9 &\quad a_{10} &\quad a_{11} &\quad a_{12} &\quad a_{13} &\quad a_{14} &\quad a_{15} \\
    a_0 &\quad a_0-1 &\quad a_2 &\quad a_{2-3} &\quad a_4 &\quad a_{4-5} &\quad a_6 &\quad a_{6-7} &\quad a_8 &\quad a_{8-9} &\quad a_{10} &\quad a_{10-11} &\quad a_4 &\quad a_{12-13} &\quad a_{14} &\quad a_{14-15} \\
    a_0 &\quad a_0-1 &\quad a_2 &\quad a_{0-3} &\quad a_4 &\quad a_{4-5} &\quad a_6 &\quad a_{4-7} &\quad a_8 &\quad a_{8-9} &\quad a_{10} &\quad a_{8-11} &\quad a_{12} &\quad a_{12-13} &\quad a_{14} &\quad a_{12-15} \\
    a_0 &\quad a_0-1 &\quad a_2 &\quad a_{0-3} &\quad a_4 &\quad a_{4-5} &\quad a_6 &\quad a_{0-7} &\quad a_8 &\quad a_{8-9} &\quad a_{10} &\quad a_{8-11} &\quad a_{12} &\quad a_{12-13} &\quad a_{14} &\quad a_{8-15} \\
    a_0 &\quad a_0-1 &\quad a_2 &\quad a_{0-3} &\quad a_4 &\quad a_{4-5} &\quad a_6 &\quad a_{0-7} &\quad a_8 &\quad a_{8-9} &\quad a_{10} &\quad a_{8-11} &\quad a_{12} &\quad a_{12-13} &\quad a_{14} &\quad a_{0-15}
\end{align*}
\]
How Would You Parallelize Scan?

• Recall the parallel reduction algorithm

<table>
<thead>
<tr>
<th>A[ ]</th>
<th>a_0</th>
<th>a_1</th>
<th>a_2</th>
<th>a_3</th>
<th>a_4</th>
<th>a_5</th>
<th>a_6</th>
<th>a_7</th>
<th>a_8</th>
<th>a_9</th>
<th>a_{10}</th>
<th>a_{11}</th>
<th>a_{12}</th>
<th>a_{13}</th>
<th>a_{14}</th>
<th>a_{15}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a_0</td>
<td>a_1</td>
<td>a_2</td>
<td>a_3</td>
<td>a_4</td>
<td>a_5</td>
<td>a_6</td>
<td>a_7</td>
<td>a_8</td>
<td>a_9</td>
<td>a_{10}</td>
<td>a_{11}</td>
<td>a_{12}</td>
<td>a_{13}</td>
<td>a_{14}</td>
<td>a_{15}</td>
</tr>
<tr>
<td></td>
<td>a_0</td>
<td>a_{0-1}</td>
<td>a_2</td>
<td>a_{2-3}</td>
<td>a_4</td>
<td>a_{4-5}</td>
<td>a_6</td>
<td>a_{6-7}</td>
<td>a_8</td>
<td>a_{8-9}</td>
<td>a_{10}</td>
<td>a_{10-11}</td>
<td>a_{12}</td>
<td>a_{12-13}</td>
<td>a_{14}</td>
<td>a_{14-15}</td>
</tr>
<tr>
<td></td>
<td>a_0</td>
<td>a_{0-1}</td>
<td>a_2</td>
<td>a_{0-3}</td>
<td>a_4</td>
<td>a_{4-5}</td>
<td>a_6</td>
<td>a_{4-7}</td>
<td>a_8</td>
<td>a_{8-9}</td>
<td>a_{10}</td>
<td>a_{8-11}</td>
<td>a_{12}</td>
<td>a_{12-13}</td>
<td>a_{14}</td>
<td>a_{12-15}</td>
</tr>
<tr>
<td></td>
<td>a_0</td>
<td>a_{0-1}</td>
<td>a_2</td>
<td>a_{0-3}</td>
<td>a_4</td>
<td>a_{4-5}</td>
<td>a_6</td>
<td>a_{0-7}</td>
<td>a_8</td>
<td>a_{8-9}</td>
<td>a_{10}</td>
<td>a_{8-11}</td>
<td>a_{12}</td>
<td>a_{12-13}</td>
<td>a_{14}</td>
<td>a_{8-15}</td>
</tr>
<tr>
<td></td>
<td>a_0</td>
<td>a_{0-1}</td>
<td>a_2</td>
<td>a_{0-3}</td>
<td>a_4</td>
<td>a_{4-5}</td>
<td>a_6</td>
<td>a_{0-7}</td>
<td>a_8</td>
<td>a_{8-9}</td>
<td>a_{10}</td>
<td>a_{8-11}</td>
<td>a_{12}</td>
<td>a_{12-13}</td>
<td>a_{14}</td>
<td>a_{0-15}</td>
</tr>
</tbody>
</table>
How Would You Parallelize Scan?

• Let \( P = \text{scan}\_\text{exclusive}(\oplus, A) \)
How Would You Parallelize Scan?

• Let \( P = \text{scan}\textunderscore\text{exclusive}(\oplus, A) \)
How Would You Parallelize Scan?

- Let \( P = \text{scan\_exclusive}(\oplus, A) \)

\[
14 = 8 + 4 + 2 \\
p_{14} = a_{0-7} + a_{8-11} + a_{12-13}
\]
How Would You Parallelize Scan?

• Let $P = \text{scan\_exclusive}(\oplus, A)$

$14 = 8 + 4 + 2$

$p_{14} = a_{0-7} + a_{8-11} + a_{12-13}$
Parallel Scan (Work Efficient Version)

- Let $P = \text{scan\_exclusive}(\oplus, A)$

$$
\begin{array}{cccccccccccccc}
A[] & a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
A'[] & a_0 & a_{0-1} & a_2 & a_{2-3} & a_4 & a_{4-5} & a_6 & a_{6-7} & a_8 & a_{8-9} & a_{10} & a_{10-11} & a_{12} & a_{12-13} & a_{14} & a_{14-15} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
A''[] & a_0 & a_{0-1} & a_2 & a_{0-3} & a_4 & a_{4-5} & a_6 & a_{4-7} & a_8 & a_{8-9} & a_{10} & a_{8-11} & a_{12} & a_{12-13} & a_{14} & a_{12-15} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
A'''[] & a_0 & a_{0-1} & a_2 & a_{0-3} & a_4 & a_{4-5} & a_6 & a_{0-7} & a_8 & a_{8-9} & a_{10} & a_{8-11} & a_{12} & a_{12-13} & a_{14} & a_{8-15} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\end{array}
$$

The Up-Sweep (Reduce) Phase
Let $P = \text{scan\_exclusive}(\oplus, A)$

**Up-Sweep Pseudocode:**

```plaintext
for d=0 to (log$_2^n$ - 1) {
    if ( tid \% (2$^{d+1}$) == 0 ) {
        a[tid + 2$^{d+1}$ -1] += a[tid + 2$^d$ -1]
    }
    barrier_synchronization();
}
```

The Up-Sweep (Reduce) Phase
Parallel Scan Code

• Let $P = \text{scan\_exclusive}(\oplus, A)$

```plaintext
A[]

\[\begin{array}{cccccccccccc}
a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15}\\
\end{array}\]
```

The Up-Sweep (Reduce) Phase

Asymptotic Complexity?
Parallel Scan Code

- Let $P = \text{scan\_exclusive}(\oplus, A)$

The Up-Sweep (Reduce) Phase
Parallel Scan (work efficient version)

- Let \( P = \text{scan\_exclusive}(\oplus, A) \)

<table>
<thead>
<tr>
<th>A'[ ]</th>
<th>( a_0 )</th>
<th>( a_{0-1} )</th>
<th>( a_2 )</th>
<th>( a_{0-3} )</th>
<th>( a_4 )</th>
<th>( a_{4-5} )</th>
<th>( a_6 )</th>
<th>( a_{0-7} )</th>
<th>( a_8 )</th>
<th>( a_{8-9} )</th>
<th>( a_{10} )</th>
<th>( a_{8-11} )</th>
<th>( a_{12} )</th>
<th>( a_{12-13} )</th>
<th>( a_{14} )</th>
<th>( a_{0-15} )</th>
</tr>
</thead>
</table>

The Down-Sweep Phase
Parallel Scan (work efficient version)

Let $P = \text{scan\_exclusive}(\oplus, A)$
Parallel Scan Code (work efficient version)

- Let \( P = \text{scan\_exclusive}(\oplus, A) \)

**Down-sweep Pseudocode:**

\[
x[n-1] \leftarrow 0
\]

for \( d = (\log_2 n - 1) \) to 0 {
   if ( \( \text{tid} \% 2^{d+1} == 0 \) ) {
      \( \text{tmp} \leftarrow a[\text{tid} + 2^d - 1] \)
      \( a[\text{tid} + 2^d - 1] \leftarrow a[\text{tid} + 2^{d+1} - 1] \)
      \( a[\text{tid} + 2^{d+1} - 1] \leftarrow \text{tmp} + a[\text{tid} + 2^{d+1} - 1] \)
   }
   \text{barrier\_synchronization}();
}
Parallel Scan Code (work efficient version)

• Let \( P = \text{scan\_exclusive}(\oplus, A) \)

\[ \begin{array}{cccccccccccccc}
A'[\ ] & a_0 & a_{0-1} & a_2 & a_{0-3} & a_4 & a_{4-5} & a_6 & a_{0-7} & a_8 & a_{8-9} & a_{10} & a_{8-11} & a_{12} & a_{12-13} & a_{14} & a_{0-15} \\
\hline
 a_0 & a_{0-1} & a_2 & a_{0-3} & a_4 & a_{4-5} & a_6 & a_{0-7} & a_8 & a_{8-9} & a_{10} & a_{8-11} & a_{12} & a_{12-13} & a_{14} & 0 \\
 a_0 & a_{0-1} & a_2 & a_{0-3} & a_4 & a_{4-5} & a_6 & a_{0-7} & a_8 & a_{8-9} & a_{10} & a_{8-11} & a_{12} & a_{12-13} & a_{14} & a_{0-7} \\
 a_0 & a_{0-1} & a_2 & 0 & a_4 & a_{4-5} & a_6 & a_{0-7} & a_8 & a_{8-9} & a_{10} & a_{8-11} & a_{12} & a_{12-13} & a_{14} & a_{0-11} \\
 a_0 & 0 & a_2 & a_{0-1} & a_4 & a_{0-3} & a_6 & a_{0-5} & a_8 & a_{0-7} & a_{10} & a_{0-9} & a_{12} & a_{0-11} & a_{14} & a_{0-13} \\
\hline
P[\ ] & 0 & a_0 & a_{0-1} & a_{0-2} & a_{0-3} & a_{0-4} & a_{0-5} & a_{0-6} & a_{0-7} & a_{0-8} & a_{0-9} & a_{0-10} & a_{0-11} & a_{0-12} & a_{0-13} & a_{0-14} \\
\end{array} \]

The Down-Sweep Phase
Parallel Scan Code (work efficient version)

- Let $P = \text{scan\_exclusive}(\oplus, A)$

<table>
<thead>
<tr>
<th>A'[ ]</th>
<th>a₀</th>
<th>a₀-1</th>
<th>a₂</th>
<th>a₀-3</th>
<th>a₄</th>
<th>a₄-5</th>
<th>a₆</th>
<th>a₀-7</th>
<th>a₈</th>
<th>a₈-9</th>
<th>a₁₀</th>
<th>a₈-11</th>
<th>a₁₂</th>
<th>a₁₂-13</th>
<th>a₁₄</th>
<th>a₀-15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₀-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₀-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₈-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₀-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₀-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₈-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>a₀-7</td>
</tr>
<tr>
<td></td>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₀-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₀-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₈-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>a₀-11</td>
</tr>
<tr>
<td></td>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>0</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₀-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₈-11</td>
<td>a₁₂</td>
<td>a₀-11</td>
<td>a₁₄</td>
<td>a₀-13</td>
</tr>
<tr>
<td></td>
<td>a₀</td>
<td>0</td>
<td>a₂</td>
<td>a₀-1</td>
<td>a₄</td>
<td>a₀-3</td>
<td>a₆</td>
<td>a₀-5</td>
<td>a₈</td>
<td>a₀-7</td>
<td>a₁₀</td>
<td>a₀-9</td>
<td>a₁₂</td>
<td>a₀-11</td>
<td>a₁₄</td>
<td>a₀-13</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>a₀</td>
<td>a₀-1</td>
<td>a₀-3</td>
<td>a₀-4</td>
<td>a₀-5</td>
<td>a₀-6</td>
<td>a₀-7</td>
<td>a₀-8</td>
<td>a₀-9</td>
<td>a₁₀</td>
<td>a₁₀-11</td>
<td>a₀-12</td>
<td>a₀-13</td>
<td>a₀-14</td>
<td></td>
</tr>
</tbody>
</table>

Asymptotic Complexity? $O(n)$

The Down-Sweep Phase
Simple Scan (when less work than parallelism)

- Let $P = \text{scan\_inclusive}(\oplus, A)$

\[
\begin{array}{cccccccccccccccc}
A'[\cdot] & a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
& a_0 & a_{0-1} & a_{1-2} & a_{2-3} & a_{3-4} & a_{4-5} & a_{5-6} & a_{6-7} & a_{7-8} & a_{8-9} & a_{9-10} & a_{10-11} & a_{11-12} & a_{12-13} & a_{13-14} & a_{14-15} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
& a_0 & a_{0-1} & a_{0-2} & a_{0-3} & a_{1-4} & a_{2-5} & a_{3-6} & a_{4-7} & a_{5-8} & a_{6-9} & a_{7-10} & a_{8-11} & a_{9-12} & a_{10-13} & a_{11-14} & a_{12-15} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
& a_{0} & a_{0-1} & a_{0-2} & a_{0-3} & a_{0-4} & a_{0-5} & a_{0-6} & a_{0-7} & a_{1-8} & a_{2-9} & a_{3-10} & a_{4-11} & a_{5-12} & a_{6-13} & a_{7-14} & a_{8-15} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
& a_0 & a_{0-1} & a_{0-2} & a_{0-3} & a_{0-4} & a_{0-5} & a_{0-6} & a_{0-7} & a_{0-8} & a_{0-9} & a_{0-10} & a_{0-11} & a_{0-12} & a_{0-13} & a_{0-14} & a_{0-15} \\
\end{array}
\]

\[
P[\cdot] & a_0 & a_{0-1} & a_{0-2} & a_{0-3} & a_{0-4} & a_{0-5} & a_{0-6} & a_{0-7} & a_{0-8} & a_{0-9} & a_{0-10} & a_{0-11} & a_{0-12} & a_{0-13} & a_{0-14} & a_{0-15} \\
\end{array}
\]

One Phase
Simple Scan (when less work than parallelism)

• Let $P = \text{scan}_{\text{inclusive}}(\oplus, A)$

\[
\begin{array}{cccccccccccccccc}
A[&] & a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
& & a_0 & a_0-1 & a_1-2 & a_2-3 & a_3-4 & a_4-5 & a_5-6 & a_6-7 & a_7-8 & a_8-9 & a_{9-10} & a_{10-11} & a_{11-12} & a_{12-13} & a_{13-14} & a_{14-15} \\
& & a_0 & a_0-1 & a_0-2 & a_0-3 & a_1-4 & a_2-5 & a_3-6 & a_4-7 & a_5-8 & a_6-9 & a_{7-10} & a_{8-11} & a_{9-12} & a_{10-13} & a_{11-14} & a_{12-15} \\
& & a_0 & a_0-1 & a_0-2 & a_0-3 & a_0-4 & a_0-5 & a_0-6 & a_0-7 & a_1-8 & a_2-9 & a_{3-10} & a_{4-11} & a_{5-12} & a_{6-13} & a_{7-14} & a_{8-15} \\
& & a_0 & a_0-1 & a_0-2 & a_0-3 & a_0-4 & a_0-5 & a_0-6 & a_0-7 & a_0-8 & a_0-9 & a_{0-10} & a_{0-11} & a_{0-12} & a_{0-13} & a_{0-14} & a_{0-15} \\
\end{array}
\]

\[
P[&] & a_0 & a_0-1 & a_0-2 & a_0-3 & a_0-4 & a_0-5 & a_0-6 & a_0-7 & a_0-8 & a_0-9 & a_{0-10} & a_{0-11} & a_{0-12} & a_{0-13} & a_{0-14} & a_{0-15} \\
\]

One Phase
Let \( P = \text{scan}_{\text{inclusive}}(\oplus, A) \)

**One-pass Simple Scan Pseudocode:**

\[
\text{for } d = 0 \text{ to } (\log_2 n - 1) \{ \\
\quad \text{if ( tid } \geq 2^{d+1} - 1) \{ \\
\quad \quad \text{tmp } \leftarrow \text{a[tid } - 2^d] + \text{a[tid]} \\
\quad \quad \text{barrier}_\text{synchronization}(); \\
\quad \text{a[tid]} = \text{tmp}; \\
\quad \text{barrier}_\text{synchronization}(); \\
\quad \} \\
\} 
\]
Simple Scan Code

• Let $P = \text{scan}_\text{inclusive}(\oplus, A)$
Simple Scan Code

- Let $P = \text{scan}_\text{inclusive}(\oplus, A)$

Asymptotic Complexity?

$O(n \cdot \log(n))$
Question

• Assume work-efficient scan is performed in shared memory

  - What are the potential approaches we can use to reduce bank conflicts?
Segmented Scan

• Let \( A = \{[a_0, a_1, a_2], [a_3, a_4], [a_5, a_6, a_7, a_8], \ldots \} \)
• Let \( \oplus = + \)
• \( \text{segmented\_scan\_inclusive}(\oplus, A) = \)
  \[
  \{[a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2], [a_3, a_3 \oplus a_4],
  \ [a_5, a_5 \oplus a_6, a_5 \oplus a_6 \oplus a_7, a_5 \oplus a_6 \oplus a_7 \oplus a_8], \ldots \}\]
Segmented Scan

• Let $P = \text{segmented\_scan\_inclusive}(\oplus, A)$

\begin{align*}
A[ & ] = \begin{array}{ccccccccccccccc}
  a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
\end{array} \\
\end{align*}

\begin{align*}
  a_0 & \rightarrow a_{0-1} & \rightarrow a_{1-2} & \rightarrow a_{2-3} & \rightarrow a_{3-4} & \rightarrow a_{4-5} & \rightarrow a_{5-6} & \rightarrow a_{6-7} & \rightarrow a_{7-8} & \rightarrow a_{8-9} & \rightarrow a_{9-10} & \rightarrow a_{10-11} & \rightarrow a_{11-12} & \rightarrow a_{12-13} & \rightarrow a_{13-14} & \rightarrow a_{14-15} \\
\end{align*}

\begin{align*}
  a_0 & \rightarrow a_{0-1} & \rightarrow a_{0-2} & \rightarrow a_{0-3} & \rightarrow a_{1-4} & \rightarrow a_{2-5} & \rightarrow a_{3-6} & \rightarrow a_{4-7} & \rightarrow a_{5-8} & \rightarrow a_{6-9} & \rightarrow a_{7-10} & \rightarrow a_{8-11} & \rightarrow a_{9-12} & \rightarrow a_{10-13} & \rightarrow a_{11-14} & \rightarrow a_{12-15} \\
\end{align*}

\begin{align*}
  a_0 & \rightarrow a_{0-1} & \rightarrow a_{0-2} & \rightarrow a_{0-3} & \rightarrow a_{0-4} & \rightarrow a_{0-5} & \rightarrow a_{0-6} & \rightarrow a_{0-7} & \rightarrow a_{1-8} & \rightarrow a_{2-9} & \rightarrow a_{3-10} & \rightarrow a_{4-11} & \rightarrow a_{5-12} & \rightarrow a_{6-13} & \rightarrow a_{7-14} & \rightarrow a_{8-15} \\
\end{align*}

\begin{align*}
  a_0 & \rightarrow a_{0-1} & \rightarrow a_{0-2} & \rightarrow a_{0-3} & \rightarrow a_{0-4} & \rightarrow a_{0-5} & \rightarrow a_{0-6} & \rightarrow a_{0-7} & \rightarrow a_{0-8} & \rightarrow a_{0-9} & \rightarrow a_{0-10} & \rightarrow a_{0-11} & \rightarrow a_{0-12} & \rightarrow a_{0-13} & \rightarrow a_{0-14} & \rightarrow a_{0-15} \\
\end{align*}

\begin{align*}
P[ & ] = \begin{array}{ccccccccccccccc}
  a_0 & a_{0-1} & a_{0-2} & a_3 & a_{3-4} & a_5 & a_{5-6} & a_{5-7} & a_{5-8} & a_9 & a_{9-10} & a_{11} & a_{11-12} & a_{11-13} & a_{11-14} & a_{11-15} \\
\end{array} \\
\end{align*}

One Phase
Segmented Scan

- Let $P = \text{segmented\_scan\_inclusive}(\oplus, A)$

A[]

| a_0 | a_1 | a_2 | a_3 | a_4 | a_5 | a_6 | a_7 | a_8 | a_9 | a_10 | a_11 | a_12 | a_13 | a_14 | a_15 |

P[]

| a_0 | a_0-1 | a_0-2 | a_3 | a_3-4 | a_5 | a_5-6 | a_5-7 | a_5-8 | a_9 | a_9-10 | a_11 | a_11-12 | a_11-13 | a_11-14 | a_11-15 |

One Phase
Segmented Scan

• Let $P = \text{segmented\_scan\_inclusive}(\oplus, A)$
Segmented Scan

- Let $P = \text{segmented\_scan\_inclusive}(\oplus, A)$
Segmented Scan

- Let \( P = \text{segmented\_scan\_inclusive}(\oplus, A) \)

\[
\begin{array}{cccccccccccccc}
A[&] & \begin{array}{cccccccccccccc}
a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
\end{array} \\
\end{array}
\]

\[
\begin{array}{cccccccccccccc}
P[&] & \begin{array}{cccccccccccccc}
a_0 & a_0-1 & a_1-2 & a_3 & a_3-4 & a_5 & a_5-6 & a_6-7 & a_7-8 & a_9 & a_9-10 & a_{11} & a_{11-12} & a_{12-13} & a_{13-14} & a_{14-15} \\
\end{array} \\
\end{array}
\]

One Phase
Segmented Scan

• Let $P = \text{segmented\_scan\_inclusive}(\oplus, A)$
Segmented Scan

• Let \( P = \text{segmented\_scan\_inclusive}(\oplus, A) \)
Segmented Scan

• Let \( P = \text{segmented\_scan\_inclusive}(\oplus, A) \)
Segmented Scan

• Let $P = \text{segmented\_scan\_inclusive}(\oplus, A)$
Let $P = \text{segmented_scan_inclusive}(\oplus, A)$
Segmented Scan Code

• Let $P = \text{segmented\_scan\_inclusive}(\oplus, A)$

**One-pass Segmented Scan Pseudocode:**

```c
for d = 0 to (\log_2 n - 1) {
    if (tid >= 2^{d+1} - 1 && flag[tid] == flag[tid - 2^d]) {
        tmp ← a[tid - 2^d] + a[tid]
    }
    barrier_synchronization();
    if (tid >= 2^{d+1} - 1 && flag[tid] == flag[tid - 2^d]) {
        a[tid] = tmp;
    }
    barrier_synchronization();
}
P = A;
```
Segmented Scan for the Work Efficient Version

- Any idea?

<table>
<thead>
<tr>
<th>A[ ]</th>
<th>a₀</th>
<th>a₁</th>
<th>a₂</th>
<th>a₃</th>
<th>a₄</th>
<th>a₅</th>
<th>a₆</th>
<th>a₇</th>
<th>a₈</th>
<th>a₉</th>
<th>a₁₀</th>
<th>a₁₁</th>
<th>a₁₂</th>
<th>a₁₃</th>
<th>a₁₄</th>
<th>a₁₅</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₂-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₆-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₁₀-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>a₁₄-15</td>
</tr>
<tr>
<td></td>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₀-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₄-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₈-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>a₁₂-15</td>
</tr>
<tr>
<td></td>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₀-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₀-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₈-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>a₈-15</td>
</tr>
<tr>
<td></td>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₀-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₀-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₈-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>a₀-15</td>
</tr>
</tbody>
</table>
Segmented Scan for the Work Efficient Version

- Any idea?

A[ ]

Up-Sweep Phase
Segmented Scan for the Work Efficient Version)

• Any idea?
**Extra-Credit Assignment**

Design your own algorithm for the segmented work-efficient scan
1. Specify the task that every thread does
2. Specify the data every thread works on
3. Is there any GPU-specific performance hazard in your algorithm?
4. If yes to question 3, how are you going to resolve it?

<table>
<thead>
<tr>
<th>a0</th>
<th>0</th>
<th>a2</th>
<th>a0-1</th>
<th>a4</th>
<th>a3</th>
<th>a6</th>
<th>a5</th>
<th>a8</th>
<th>a5-7</th>
<th>a10</th>
<th>a9</th>
<th>a12</th>
<th>a11</th>
<th>a14</th>
<th>a11-13</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[0]</td>
<td>0</td>
<td>a0</td>
<td>a0-1</td>
<td>0</td>
<td>a3</td>
<td>0</td>
<td>a5</td>
<td>a5-6</td>
<td>a5-7</td>
<td>0</td>
<td>a9</td>
<td>0</td>
<td>a11</td>
<td>a11-12</td>
<td>a11-13</td>
</tr>
</tbody>
</table>

The Down-Sweep Phase
Hierarchical Scan

- Perform Scan in Local Chunk of Data first and Merge the Results later

64-element scan
local_scan_1

64-element scan
local_scan_2

64-element scan
local_scan_3

64-element scan
local_scan_4

Get the last element of each local scan

Perform scan again

Add base to all elements in each local scan

Add base to
local_scan_2

Add base to
local_scan_3

Add base to
local_scan_32
Hierarchical Segmented Scan

- Perform Scan in Local Chunk of Data first and Merge the Results later

64-element scan
local_seg_scan_1

64-element scan
local_seg_scan_2

64-element scan
local_seg_scan_3

64-element scan
local_seg_scan_4

Get the last element of each local segmented scan

Perform segmented scan again

If the base is in the same segment as the element in local segmented scan, add the base to the first segment in each local result
An Example of Parallel Segmented Scan

• Sparse Matrix Vector Multiplication (SpMV)

Let $A = \begin{bmatrix} 3*x_0, & x_2, & [2*x_1], & [4*x_2] \\ 2*x_1, & 6*x_2, & \ldots, & 8*x_{n-1} \end{bmatrix}$, and $\oplus = +$
for SpMV, $y = \text{segmented\_scan}(\oplus, A)$

• Project 1 requires a segmented scan at the warp level
Lecture 3B: Lexical and Syntax Analysis I

(The lectures are based on the slides copyrighted by Keith Cooper and Linda Torczon from Rice University.)
Scanner and Parser collaborate to check the syntax of the input program.

1. Scanner maps stream of characters into words (words are the fundamental unit of syntax)

Scanner produces a stream of tokens for the parser

Token is <part of speech, word>

“Part of speech” is a unit in the grammar

2. Parser maps stream of words into a sentence in a grammatical model of the input language
Review — Three Pass Compiler

Scanner and Parser collaborate to check the syntax of the input program.

1. Scanner maps stream of characters into words (words are the fundamental unit of syntax)

Scanner produces a stream of tokens for the parser

Token is <part of speech, word>

“Part of speech” is a unit in the grammar

2. Parser maps stream of words into a sentence in a grammatical model of the input language
The Compiler’s Front End

- **Scanner** looks at every character
  - Converts stream of characters to stream of tokens, or classified words:
    - `<part of speech, word>`
  - Efficiency & scalability matter
    - Scanner is only part of compiler that looks at every character

- **Parser** looks at every token
  - Determines if the stream of tokens forms a sentence in the source language
  - Fits tokens to some syntactic model, or grammar, for the source language
The Compiler’s Front End

• **Scanner** looks at every character
  - Converts stream of characters to stream of tokens, or classified words:
    - `<part of speech, word>`
  - Efficiency & scalability matter
    Scanner is only part of compiler that looks at every character

• ** Parser** looks at every token
  - Determines if the stream of tokens forms a sentence in the source language
  - Fits tokens to some syntactic model, or grammar, for the source language
How can we automate the construction of scanner & parsers?

• **Scanner**
  - Specify syntax with regular expressions (REs)
  - Construct finite automaton & scanner from the regular expression

• **Parser**
  - Specify syntax with context-free grammars (CFGs)
  - Construct push-down automaton & parser from the CFG
Suppose that we need to recognize the word “not”:

How would we look for it?
- Calling some routine?
  - for example, fscanf(), the Scanner class in Java, or a regex library (Python, Java, C++)
- In a compiler, we would build a recognizer (not just calling a routine)

Implementation idea
- The spelling is ’n’ followed by ‘o’ followed by ’t’
- The code shown to the right, is as simple as the description
  - Cost is O(1) per character
  - Structure allows for precise error messages

Pseudocode:

```plaintext
c ← next character
If c = ‘n’ then {
  c ← next character
  if c = ‘o’ then {
    c ← next character
    if c = ‘t’
      then return <NOT,"not">
    else report error
  }
  else report error
}
else report error
```
Automata

We can represent this code as an automaton

c ← next character
If c = ‘n’ then {
    c ← next character
    if c = ‘o’ then {
        c ← next character
        if c = ‘t’
            then return <NOT,”not”>
        else report error
    }
    else report error
} else report error

Transition Diagram for “not”

States drawn with double lines indicate success
Automata

To recognize a larger set of keywords is (relatively) easy

For example, the set of words \{ if, int, not, then, to, write \}

one-to-one map from final states to words (maybe parts of speech)

Cost is still \( O(1) \) per character
Specifying An Automaton

We list the words
Each word’s spelling is unique and concise
We can separate them with the | symbol, read as “or”

The specification:

if int not then to write

is a simple regular expression for the language accepted by our automaton
Specifying An Automaton

• Every Regular Expression corresponds to an automaton
- We can construct, automatically, an automaton from the RE
- We can construct, automatically, an RE from any automaton
- We can convert an automaton easily and directly into code
- The automaton and the code both have $O(1)$ cost per input character

An RE specification leads directly to an efficient scanner
Unsigned Integers

In the example, each word had one spelling

What about a single part of speech with many spellings?

Consider specifying an unsigned integer
  • An unsigned integer is any string of digits from the set [0…9]
  • We might want to specify that no leading zeros are allowed

is a simple regular expression for the language accepted by our automaton

The Automata

With Leading Zeros

Without Leading Zeros

How do we write the corresponding RE?

Is “0001” allowed? Or “00”?
Unsigned Integers

• The Automata

- We need a notation to represent that cyclic edge in the automata
- The Kleene Closure
  \( x^* \) represents zero or more instances of ‘x’
- The Positive Closure
  \( x^+ \) represents one or more instances of ‘x’

\([0-9]^+\) for leading zeros; 0 l \([1-9] [0-9]^*\) for without leading zeros
Unsigned Integers

- The RE corresponds to an automaton and an implementation

\[ \begin{align*}
&c \leftarrow \text{next character} \\
&n \leftarrow 0 \\
&\text{if } c = '0' \text{ then return } <\text{CONSTANT},n> \\
&\text{else if } ('1' \leq c \leq '9') \text{ then } \{ \\
&\quad n \leftarrow \text{atoi}(c) \\
&\quad c \leftarrow \text{next character} \\
&\quad \text{while } ('0' \leq c \leq '9') \{ \\
&\quad\quad t \leftarrow \text{atoi}(c) \\
&\quad\quad n \leftarrow n * 10 + t \\
&\quad\} \\
&\quad \text{return } <\text{CONSTANT},n> \\
&\} \\
&\text{else report error}
\end{align*} \]

The automaton and the code can be generated automatically
Regular Expression

• A formal definition

Regular Expressions over an Alphabet $\Sigma$

If $x \in \Sigma$, then $x$ is an RE denoting the set $\{ x \}$ or the language $L = \{ x \}$
If $x$ and $y$ are REs then

$\text{xy}$ is an RE denoting $L(x)L(y) = \{ pq \mid p \in L(x) \text{ and } q \in L(y) \}$

$x \cup y$ is an RE denoting $L(x) \cup L(y)$

$x^*$ is an RE denoting $L(x)^* = \bigcup_{0 \leq k < \infty} L(x)^k$  \hspace{1cm} (Kleene Closure)

Set of all strings that are zero or more concatenations of $x$

$x^+$ is an RE denoting $L(x)^+ = \bigcup_{1 \leq k < \infty} L(x)^k$  \hspace{1cm} (Positive Closure)

Set of all strings that are one or more concatenations of $x$

$\varepsilon$ is an RE denoting the empty set
Regular Expression

- How do these operators help?

Regular Expressions over an Alphabet $\Sigma$

If $x$ is in $\Sigma$, then $x$ is an RE denoting the set $\{ x \}$ or the language $L = \{ x \}$

*The spelling of any letter in the alphabet is an RE*

If $x$ and $y$ are REs then

- $xy$ is an RE denoting $L(x)L(y) = \{ pq \mid p \in L(x) \text{ and } q \in L(y) \}$
  
  *If we concatenate letters, the result is an RE (spelling of words)*

- $x \mid y$ is an RE denoting $L(x) \cup L(y)$
  
  *Any finite list of words can be written as an RE, ( $w_0 \mid w_1 \mid w_2 \mid \ldots \mid w_n$ )*

- $x^*$ is an RE denoting $L(x)^* = \bigcup_{0 \leq k < \infty} L(x)^k$

- $x^+$ is an RE denoting $L(x)^+ = \bigcup_{1 \leq k < \infty} L(x)^k$

*We can use closure to write finite descriptions of infinite, but countable, sets*

$\varepsilon$ is an RE denoting the empty set

*In practice, the empty string is often useful*
Regular Expression

- Let the notation [0-9] be shorthand for (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)

RE Examples:

Non-negative integer  
\[ [0-9] [0-9]^* \]

or  
\[ [0-9]^* \]

No leading zeros  
\[ 0 | [1-9] [0-9]^* \]

Algol-style Identifier  
\[ ([a-z]|[A-Z]) ([a-z]|[A-Z]|[0-9])* \]

Decimal number  
\[ 0 | [1-9] [0-9]^* . [0-9]^* \]

Real number  
\[ ((0 | [1-9] [0-9]^*) | (0 | [1-9] [0-9]^* . [0-9]^*) \ E [0-9] [0-9]^* \]

Each of these REs corresponds to an automaton and an implementation.
From RE to Scanner

• We can use results from automata theory to construct scanners directly from REs

- There are several ways to perform this construction
- Classic approach is a two-step method.
  - 1. Build automata for each piece of the RE using a simple template-driven method
    Build a specific variation on an automaton that has transitions on ε and non-deterministic choice (multiple transitions from a state on the same symbol)
    This construction is called “Thompson’s construction”
  - 2. Convert the newly built automaton into a deterministic automaton
    Deterministic automaton has no ε-transitions and all choices are single-valued
    This construction is called the “subset construction”

- Given the deterministic automaton, we can run a minimization algorithm on it to reduce the number of states
  - Minimization is a space optimization. Both the original automaton and the minimal one take $O(1)$ time per character
Thompson’s Construction

• The Key Idea

- For each RE symbols and operator, we have a small template
- Build them, in precedence order, and join them with $\varepsilon$-transitions

Precedence in REs: Parentheses, then closure, then concatenation, then alternation
Thompson’s Construction

• Let’s build an NFA for \( a (b \mid c)^* \)

1. \( a, b, \) & \( c \)

2. \( b \mid c \)

3. \((b \mid c)’\)

4. \( a (b \mid c)’\)

\[ \text{Precedence in REs: Parentheses, then closure, then concatenation, then alternation} \]
Subset Construction

• The Concept

- Build a simpler automaton (no $\varepsilon$-transitions, no multi-valued transitions) that simulates the behavior of the more complex automaton
- Each state in the new automaton represents a set of states in the original

### DFA

$$a (b \mid c)^*$$

### NFA

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0$</td>
<td>$n_0$</td>
</tr>
<tr>
<td>$d_1$</td>
<td>$n_1 \ n_2 \ n_3 \ n_4 \ n_6 \ n_9$</td>
</tr>
<tr>
<td>$d_2$</td>
<td>$n_5 \ n_8 \ n_9 \ n_3 \ n_4 \ n_6$</td>
</tr>
<tr>
<td>$d_3$</td>
<td>$n_7 \ n_8 \ n_9 \ n_3 \ n_4 \ n_6$</td>
</tr>
</tbody>
</table>
Minimization

- DFA minimization algorithms work by discovering states that are equivalent in their contexts and replacing multiple states with a single one.

- Minimization reduces the number of states, but does not change the costs.

<table>
<thead>
<tr>
<th>Minimal DFA State</th>
<th>$s_0$</th>
<th>$s_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original DFA State</td>
<td>$d_0$</td>
<td>$d_1, d_2, d_3$</td>
</tr>
</tbody>
</table>
Implementing an Automaton

• A common strategy is to simulate the DFA’s execution
  
  - Skeleton parser + a table that encodes the automaton
  - The scanner generator constructs the table
  - The skeleton parser does not change (much)

```plaintext
state ← s_0
char ← NextChar()
while (char ≠ EOF) {
    state ← δ[state,char]
    char ← NextChar()
}
if (state is a final state)
    then report success
else report an error
```

Simple Skeleton Scanner

Transition table for our minimal DFA

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_0</td>
<td>S_1</td>
<td>S_e</td>
<td>S_e</td>
</tr>
<tr>
<td>S_1</td>
<td>S_e</td>
<td>S_1</td>
<td>S_1</td>
</tr>
</tbody>
</table>
Automatic Scanner Construction

- **Scanner Generator**
  - Tasks in a specification written as a collection of regular expressions
  - Combines them into one RE using alternation ("|")
  - Builds the minimal automaton (§ 2.4, 2.6.2 EAC)
  - Emits the tables to drive a skeleton scanner (§ 2.5 EAC)
Automatic Scanner Construction

- **Scanner Generator**
  - As alternative, the generator can produce code rather than tables
  - Direct-coded scanners are ugly, but often faster table-driven scanners
  - Other than speed, the two are equivalent
Summary

• **Automating Scanner Construction**
  - Write down the REs for the input language and connect them with “|”
  - Build a big, simple NFA
  - Build the DFA that simulates the NFA
  - Minimize the number of states in the DFA
  - Generate an implementation from the minimal DFA

• **Scanner Generators**
  - lex, flex, jflex, and such all work along the lines
  - Algorithms are well-known and well-understood
  - Key issues are: finding longest match rather than first match, and engineering the interface to the parser
  - You can build your own scanner generator

“Find the longest match” is where $O(1)$ per character can turn into $O(n)$. More next lecture.
Reading

- “Parallel Prefix Sum (Scan) with CUDA”, GPU Gem3, Chapter 39, by Mark Harris, Shubhabrata Sengupta, and John Owens.
- “Performance Optimization” by Paulius Micikevicius, NVIDIA
- CUDA C/C++ Basics
- Loop unrolling for GPU Computing
- CUDA Programming Guide, Chapter 1-5.
- Book: Engineering a Compiler (EAC)
  - Chapter 2.2
  - Chapter 2.4
  - Chapter 2.5
  - Chapter 2.6