CSC266 Introduction to Parallel Computing using GPUs
Parallelizing Programs

Sreepathi Pai
September 20, 2017
URCS
Outline

- Dependences
- Important Archetypes
- Parallelism in Action
- Short Vector Machines
Outline

Dependences

Important Archetypes

Parallelism in Action

Short Vector Machines
What makes some algorithms serial and others parallel?
Operation waits for data to become available:

\[ a = b + c \]

which gets compiled down to:

\[
\begin{align*}
\text{load r1, b} \\
\text{load r2, c} \\
\text{add r3, r1, r2}
\end{align*}
\]

Although add can start before the two loads, it has to wait for the loads to finish.
Control Dependences

- Operation may or may not execute depending on condition

```python
if(temperature == cold):
    wear_jacket()
```

- Control dependences can always be converted to data dependences
Dynamic vs Static Dependences

When can the two writes in the code below execute in parallel?

```c
void set_p(int *x, int *y) {
    *x = 1;
    *y = 2;
}
```

When can they not?
Dependence Graphs

- A graphical representation of dependences between operations
  - Data-flow graphs
  - Control-flow graphs
- We will focus on data-flow graphs
  - Nodes: operations in dynamic trace
  - Edges: communication, “flow of data”
- Dynamic trace
  - All operations executed by a program other than control flow
Assume \( a, b \) and \( c \) are not aliases (i.e. they are separate arrays) and \( N = 3 \).

\[
\text{for}(i = 0; i < N; i++) \\
\quad a[i] = b[i] + c[i]
\]

The dynamic trace is:

\[
\begin{align*}
    a[0] &= b[0] + c[0] \\
\end{align*}
\]
$$a[0] = b[0] + c[0]$$
Outline

Dependences

Important Archetypes

Parallelism in Action

Short Vector Machines
Assume a, b and c all arrays of length $N$.

```c
void vadd(a, b, c, N) {
    for(i = 0; i < N; i++)
        a[i] = b[i] + c[i]
}
```

Does iteration 1 of this loop depend on iteration 0 of this loop?
Dependence Graph for Vector Addition

\[
\]
Distributing Work for Vector Addition

Sometimes called “embarrassingly parallel” – no communication between parallel computations
Assume \( a \) is array of length \( N = 3 \).

```c
void vsum(a) {
    S = 0
    for(i = 0; i < N; i++)
        S = S + a[i]
}
```

What does the dependence graph for this loop look like?
\[ S = 0 \]
\[ S = S + A[0] \]
\[ S = S + A[1] \]
• Create an array of $S$
  • One element for each thread
• Get sum of each thread as an array
  • $S[0..thread]$
• Repeat on this array until only one element left
• Requires synchronization
Distributing Work for Vector Sum

\[ a[2] + 0 \rightarrow S1 \]

\[ a[0] + a[1] \rightarrow S0 \]

CPU1

CPU0

Barrier

\[ + \rightarrow S \]

CPU0
Nested Parallelism: Matrix Multiplication

- Multiply matrices $A$ and $B$ to get $C$
  - $A$ is $M \times N$
  - $B$ is $N \times K$
  - $C$ is $M \times K$

```c
for(row = 0; row < M; row++)
  for(col = 0; col < K; col++)
    vmul(A[row][:], B[:][col], Tmp, N)
    C[row][col] = vsum(Tmp, N)
```
Which of the loops below can be safely parallelized?

```c
void bfs(graph, LEVEL) {
    for(n = 0; n < graph.nodes; n++)
        for(e = n.firstedge(); e < n.lastedge(); e++)
            if(e.dst.distance == INF && n.distance == LEVEL - 1)
                e.dst.distance = LEVEL;
}

set all distances to INF
src.distance = 0
LEVEL = 1
do {
    bfs(graph, LEVEL)
    LEVEL++
} while(some edge’s distance changed)
```
BFS on two graphs

Does parallelism depend on input?
Flynn’s Taxonomy

How computers are implemented:

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)
Programmer’s Viewpoint

- Task Parallelism
- Data Parallelism
Task Parallelism

• Different code (usually all running at same time)
• Different data
• Example (real life)
  • Assembly line
• Example (CS)
  • Unix pipe invocation: `sort | uniq | wc -l`
Data Parallelism

- Same code
- Runs on different data
- Two important levels
  - Instruction-level (SIMD)
  - Program-level (SPMD)

GPUs prefer data parallelism.
SIMD Implementations: Vector Machines

The Cray-1 (circa 1977):

- \( V_x \) – vector registers
  - 64 elements
  - 64-bits per element
- Vector length register (\( Vlen \))
- Vector mask register

For $0 < i < Vlen$:

$$dst[i] = src1[i] + src2[i]$$

- Most arithmetic instructions
Vector Instructions – Horizontal

\[ 1 = \min(1, 2, 3, 4) \]

For \(0 < i < V\text{len} \):

\[ \text{dst} = \min(\text{src1}[i], \text{dst}) \]

Note that \(\text{dst}\) is a scalar.

- Mostly reductions (min, max, sum, etc.)
- Not well supported
  - Cray-1 did not have this
Vector Instructions – Shuffle/Permute

\[
\begin{array}{c}
\text{src} \\
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array} \\
\text{mask} \\
\begin{array}{cccc}
0 & 3 & 1 & 1 \\
\end{array} \\
\text{dst} \\
\begin{array}{cccc}
1 & 4 & 2 & 2 \\
\end{array}
\end{array}
\]

dst = shuffle(src1, mask)

- Poor support on older implementations
- Reasonably well-supported on recent implementations
g5mask = gt(src1, 5)
dst = mul(src1, src2, g5mask)
Outline

Dependences

Important Archetypes

Parallelism in Action

Short Vector Machines
SIMD Registers in modern CPUs

- Bits, not elements
  - 64 bits (MMX, 3DNow!)
  - 128 bits (SSE family, AltiVec)
  - 256 bits (AVX-256)
  - 512 bits (AVX 2.0 or AVX-512)

- Example 128-bit vector can contain
  - 16 8-bit elements
  - 8 16-bit elements
  - 4 32-bit elements
  - 2 64-bit elements
  - 1 128-bit element

- No vector length register
  - except AVX512 which has vector length (but can only choose 128, 256, 512)

- No vector masks
  - except AVX512 which has masks
Four ways to write SIMD programs on current CPUs

In order of decreasing difficulty:

- Assembly code
  - not covering this
- SIMD intrinsics
- Vector types
- Autovectorizing compilers
  - not covering this
  - great when it works!
Vector intrinsics (using x86 as an example)

- Expose short vector instructions to programmer
- Not assembly programming, compiler still does:
  - register allocation
  - scheduling
- New data types (like vector types):
  - _m64 (MMX)
  - _m128 (SSE)
  - _m128i and _m128d (SSE2)
- Intrinsics look like functions:
  - _m128 _mm_add_ss (_m128 A, _m128 B)
- Allows direct use of machine instructions
typedef int v4si __attribute__ ((vector_size (16)));

v4si a, b, c;

c = a + b;

- Operations on vector types generate vector instructions
  - Most low-level details (e.g. alignment), taken care of by gcc
- Operations supported:
  - +, -, *, /, unary minus, ^, |, &, ~, %
  - this does not include machine-specific instructions
• Dependence analysis is necessary to identify parallelism

• Important types of parallelism “patterns”:
  • embarrassingly parallel (vector addition)
  • tree-based reductions
  • nested parallelism
  • irregular parallelism

• For GPUs: focus on SIMD parallelism
  • Abundant in scientific applications, multimedia applications, etc.
  • Usually found in loops