class 10

**fast scans 2.0**

prof. Stratos Idreos

HTTP://DASLAB.SEAS.HARVARD.EDU/CLASSES/CS165/
always want to minimize data movement - computation

& utilize all resources!
shared scans

q1 q2 q3 q4 q5 q6 ... ... qn
You control how data moves by what you are trying to “touch” through your algorithms.

Ideal cost: $O(N)$ pages, $N$ is the number of pages of column
for (j = 0; j < n; j += vectorsize)

Q1: for (i = j; i < j + vectorsize; i++) {
    min = a[i] < min ? a[i] : min
}

Q2: for (i = j; i < j + vectorsize; i++) {
    max = a[i] > max ? a[i] : max
}

... for (j = 0; j < n; j += vectorsize)

Q1: for (i = j; i < j + vectorsize; i++) {
    min = a[i] < min ? a[i] : min
}

Q2: for (i = j; i < j + vectorsize; i++) {
    max = a[i] > max ? a[i] : max
}

...

Thread 1
Pin to Core 1

You control how data moves by what you are trying to “touch” through your algorithms.

Ideal cost: O(N) pages, N is the #pages of column
1. can do >1 tasks at the same time

2. can predict data accesses & execute instructions out of order

3. can do the same task on >1 data items
RISC CPU architecture

we can “synthesize” all instructions out of a small set of “hardware primitives”
RISC CPU architecture

e.g., res = a + b
1) load a
2) load b
3) r = a + b

fetch instruction 1),
decode,
execute (find memory location),
read from memory,
write to register
pipelining
>1 instructions at a time
pipelining
>1 instructions at a time
pipelining
>1 instructions at a time

parallel execution
**Pipelining**

>1 instructions at a time

Parallel execution

Ideally execution is $P$ times faster where $P$ is the depth of the pipeline
next instruction starts before the previous one finishes

but which one is next?
CPU

fetch | decode | execute | mem | write
fetch | decode | execute | mem | write
fetch | decode | execute | mem | write

... if (X)
  instr.A1
  instr.A2
...

... instr.B1
  instr.B2
...

... a=k*z
  r=a+j
...

dependencies
CPU

fetch | decode | execute | mem | write
---|---|---|---|---
fetch | decode | execute | mem | write
fetch | decode | execute | mem | write
fetch | decode | execute | mem | write

... next instruction starts before the previous one finishes

... which one is next?

... if (X)
    instr.A1
    instr.A2
    ...
    instr.B1
    instr.B2
    ...

... a=k*z
    r=a+j
    ...

dependencies

class speculative execution
speculative execution

CPU may stall or even have to flush pipeline

dependencies

if (X)
  instr.A1
  instr.A2
  ...
  instr.B1
  instr.B2
  ...
  ...
a = k * z
r = a + j
**avoid**

dependencies and branches whenever possible

i instructions

**for** all tuples

do { ... }

j instructions
`count(A<v)`

```
for(i=0;i<A.size;i++)
    if A[i]<v
        res++
```

what is the response time behavior depending on selectivity?

Selectivity affects how many times CPU should pick instructions from “inside or outside” the if statement.
for(i=0;i<A.size;i++)
    if A[i]<v
        res++

you would think…
for(i=0;i<A.size;i++)
    if A[i]<v
        res++

# of tuples that qualify
0                   N
response time

you would think...

instead

# of tuples that qualify
0                   N
response time
for(i=0;i<A.size;i++)
    if A[i]<v
        res++

you would think…

because

instead
more is sometimes less...

(1) **with branches**
   for(i=0;i<N;i++)
       if (A[i]<v)
           result++

(2) **no branches**
   for(i=0;i<N;i++)
       result+=(A[i]<v)
(1) **using &&**
   
   ```
   for(i=0;i<N;i++)
     if (f1(a1) && f2(a2) && ... && fk(ak))
       result[j++] = i
   ```

(2) **using &**
   
   ```
   for(i=0;i<N;i++)
     if (f1(a1) & f2(a2) & ... & fk(ak))
       result[j++] = i
   ```

(3) **no branches**
   
   ```
   for(i=0;i<N;i++)
     result[j] = i
     j += (f1(a1) & f2(a2) & ... & fk(ak))
   ```
(1) **with branches**
   ```cpp
   for(i=0;i<N;i++)
     if (A[i]<v)
       result++
   ```

(2) **no branches**
   ```cpp
   for(i=0;i<N;i++)
     result+=(A[i]<v)
   ```

(3) **no branches**
   ```cpp
   for(i=0;i<N;i+=2)
     result1+=(A[i]<v)
     result2+=(A[i+1]<v)
   result=result1+result2
   ```
prefetching
predict future data accesses

CPU
registers
on chip cache
on board cache
memory
disk
to process data X we need to fetch X

fetch

prefetching:
while processing Z fetch X on last level cache

hardware assisted +
software assisted

scan vs index scan
1. can do >1 tasks at the same time

2. can predict data accesses & execute instructions out of order

3. can do the same task on >1 data items
apply the same instruction on >1 data values with the same cost \textbf{SIMD} (single instruction multiple data) as for 1 value

\begin{center}
\begin{tabular}{cccc}
\textbf{result:} 128 bit & & & \\
res1, res2, res3, res4 & & & \\
\hline
\end{tabular}
\end{center}

run at the same time \begin{tabular}{cccc}
op & op & op & op \\
\end{tabular}

A1, A2, A3, A4 \quad B1, B2, B3, B4

\begin{tabular}{c}
\textbf{input 1:} 128 bit \\
\end{tabular} \quad \begin{tabular}{c}
\textbf{input2:} 128 bit \\
\end{tabular}
apply the same instruction on >1 data values with the same cost

**SIMD** (single instruction multiple data) as for 1 value

- **input 1**: 128 bit
- **input 2**: 128 bit
- **result**: 128 bit

- **A1, A2, A3, A4**
- **B1, B2, B3, B4**

run at the same time
apply the same instruction on >1 data values with the same cost

**SIMD** (single instruction multiple data) as for 1 value

run at the same time

```
result: 128 bit
res1, res2, res3, res4

op  op  op  op

A1, A2, A3, A4
input 1: 128 bit

B1, B2, B3, B4
input2: 128 bit
```
apply the same instruction on >1 data values with the same cost \textbf{SIMD (single instruction multiple data)} as for 1 value

run at the same time

\begin{itemize}
  \item \textbf{result:} 128 bit
  \item \textbf{input 1:} 128 bit
  \item \textbf{input 2:} 128 bit
  \item \textbf{A1, A2, A3, A4}
  \item \textbf{B1, B2, B3, B4}
  \item \textbf{res1, res2, res3, res4}
\end{itemize}
apply the same instruction on >1 data values with the same cost. **SIMD** (single instruction multiple data) as for 1 value.
result: 128 bit

input 1: 128 bit

input2: 128 bit

6, 13, 13, 4

5, 4, 7, 2

1, 9, 6, 2
result: 128 bit

0, 4294967295, 0, 4294967295

input 1: 128 bit

input2: 128 bit
**how**: assembly || compilers || SIMD intrinsics

>1 assembly instructions
write two sum operator using SIMD intrinsics

\[
\text{for (i=0; i<\text{data.size}; i++)} \\
\text{res += data[i]}
\]

for (i=0; i<\text{data.size}; i++)
if data[i]<v
res += data[i]

you have the following SIMD intrinsics:
- **SIMD\_add**: \([A1,B1,C1,D1], [A2,B2,C2,D2] \rightarrow [A1+\text{A2}, B1+B2, C1+C2, D1+D2]\)
- **SIMD\_shuffle32**: \([A,B,C,D] \rightarrow [B,A,D,C]\)
- **SIMD\_shuffle64**: \([A,B,C,D] \rightarrow [C,D,A,B]\)
- **SIMD\_and**: \([1,0,1,0] \& \ [1,1,1,1] \rightarrow [1,0,1,0]\)
- **SIMD\_lt**: \([10,2,4,10] < [4,7,3,6] \rightarrow [0,1,0,0]\)

SIMD vector=128bit
Data is 32 bit ints
for (i=0; i< data.size; i++)
    res += data[i]

for (i=0; i< N; i+=4)
    res[0,1,2,3] = SIMD_add(res[0,1,2,3], data[i,i+1,i+2,i+3])
    t1 = SIMD_shuffle32(res)
    t2 = SIMD_add(res, t1)
    t3 = SIMD_shuffle64(t2)
    res = SIMD_add(t2, t3)

res = a, b, c, d
    t1 = b, a, d, c
    t2 = a+b, a+b, c+d, c+d
    t3 = c+d, c+d, a+b, a+b
    res = a+b+c+d, a+b+c+d, a+b+c+d, a+b+c+d

(assuming N mod 4 = 0)
for (i=0; i<N; i+=4) 
  t1 = SIMD_lt(data[i,i+1,i+2,i+3], v[v,v,v,v]) 
  t2 = SIMD_and(t1[0,1,2,3], data[i,i+1,i+2,i+3]) 
  res[0,1,2,3] = SIMD_add(res[0,1,2,3], t2[0,1,2,3])

(assuming N mod 4 = 0)
column-store storage better for SIMD
data is already packed in dense arrays
Read: Selection conditions in main memory
Kenneth A. Ross
ACM Transactions on Database Systems, 2004

Browse: Implementing database operations using SIMD instructions
by Jingren Zhou, and Kenneth A. Ross
ACM SIGMOD International Conference on Management of Data, 2002

Browse: Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture
by Jatin Chhugani et al.
International Conference on Very Large Databases (VLDB), 2008
fast scans 2.0

DATA SYSTEMS

prof. Stratos Idreos