BY THEIR FRUITS SHALL YE KNOW THEM
A DATA ANALYST’S PERSPECTIVE ON MASSIVELY PARALLEL SYSTEM DESIGN

Holger Pirk       Sam Madden       Mike Stonebraker
A CRUCIAL DISTINCTION

* ISTC BIG DATA ≠ intel®
INSPIRATION
MY PLEDGE OF LOYALTY
SCIENTIFIC RATIONALE
GENE AMDAHL TAUGHT US THAT SYSTEMS NEED TO BE BALANCED

\[ \text{Processed Instructions per Second} \]

\[ \text{Processed Bytes per Instruction} \]

\[ \text{Processing 500GB/s} \]
NVIDIA AND AMD PROCESS LOT OF SMALL DATA WORDS

Processed Instructions per Second

Processed Bytes per Instruction

Processing 500GB/s
SIMT 

SIMT Cores

Instruction Scheduler

Memory
INTEL PROCESSES FEWER LARGE DATAWORDS

Processed Instructions per Second

Processed Bytes per Instruction

Processing 500GB/s
MANY-CORE SIMD

SIMD Core

SIMD Core

SIMD Core

SIMD Core

SIMD Core

Pentium Cores

512 Bits

Memory
SIMD WITH SCATTER/GATHER
ALL OF THEM CAN PROCESS WAY MORE DATA THAN THEY CAN LOAD.
SPEC BANDWIDTH-WISE, PHI OUTPERFORMS CURRENT GPUs

GB/s Memory Bandwidth

Phi

GTX 780
Our question: Does it matter? Does PHI change anything?

Processed Instructions per Second

Processed Bytes per Instruction

Processing 500GB/s

Nvidia

AMD

Intel
THE OBSTACLE COURSE
DATA-CENTRIC APPLICATIONS HAVE TYPICAL CHOKEPOINT

- Computation
- Synchronization
- Bandwidth
- Capacity

Facts
Dimension
DATA-CENTRIC APPLICATIONS HAVE TYPICAL CHOKEPOINTS

Hash Complexity → # of Conflicts
Tuple Width

Access Locality

Facts
Dimension
PHI VS. GTX 780
FIRST CHOKEPOINT

Diagram:
- Bandwidth
- Facts
- Dimension

Legend:
- π
- Γ
BANDWIDTH OF PHI LOOKS SIMILAR TO GPU AT FIRST GLANCE
A SECOND GLANCE REVEALS SOMETHING ODD…

A Non-Linear Cost Function
A SECOND GLANCE REVEALS SOMETHING ODD…

Not Dominated (only) by Cache Misses
SECOND CHOKEPOINT

Diagram:

- **Capacity**
  - Connected to a node labeled with a circle and a cross.
  - The circle and cross are connected to:
    - A node labeled with a circle and a cross.
    - A node labeled with a circle.
- **Facts**
  - Connected to the node labeled with a circle and a cross.
- **Dimension**
  - Connected to the node labeled with a circle and a cross.

Symbols used:
- π
- Γ
PHI BENEFITS FROM LARGER CACHES

Size of Lookup Table in Bytes

Time per Access in ns

GTX 780

Xeon Phi

Xeon Phi Lower Bound

GTX 780 Lower Bound
THIRD CHOKEPOINT

Computation

\( \Pi \)

Facts

Dimension
COMPUTATION PERFORMANCE IS VERY SIMILAR…

![Graph showing computation performance comparison between Xeon Phi and GTX 780.](graph.png)

- Time per hash in ns
- Number of Murmur Rehashes

0.05, 0.10, 0.20, 0.40, 0.80

1, 2, 4, 8, 16, 32

- Xeon Phi
- GTX 780
THIRD CHOKEPOINT

- π
- Γ

Synchronization

- Facts
- Dimension
... AND SO IS HASH-BUILDING
RECAP

• Phi & GPU mostly en par in
  • Computation
  • Synchronization
  • Cache-Utilization

• But what is up with the memory access
PHI IN DEPTH
SCATTER/GATHER
CHAPTER 6. INSTRUCTION DESCRIPTIONS

VGATHERDPD - Gather Float64 Vector With Signed Dword Indices

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVEX.512.66.0F38.W1 92 /r /vsib vgatherdpd ymm1 {k1}</td>
<td>Gather float64 vector (vmt) into float64 vector ymm1 using doubleword indices and k1 as completion mask.</td>
</tr>
</tbody>
</table>

Description

A set of 8 memory locations pointed by base address BASE_ADDR and doubleword index vector VINDEX with scale SCALE are converted to a float64 vector. The result is written into float64 vector ymm1.

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function SELECT_SUBSET). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, at least one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e., all elements of the gather/scatter sequence have been loaded/stored and hence, the write-mask bits are zero).

Note that accessed element by vmt always access 64 bytes of memory. The memory region accessed by each element will always lie between element_linear_address & (~0x3F) and (element_linear_address & (~0x3F)) + 63 boundaries.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element before up-conversion.

Note also the special mask behavior as the corresponding bits in write mask k1 are reset with each destination element being updated according to the subset of write mask k1. This is useful to allow conditional re-trigger of the instruction until all the elements from a given write mask have been successfully loaded.

The instruction will #GP fault if the destination vector ymm1 is the same as index vector VINDEX.

Operation

// instruction works over a subset of the write mask
kmemp = SELECT_SUBSET(k1)
// Use vmt as vector memory operand (VMEK)
for (n = 0; n < 8; n++) {
  if (kmemp[n] == 1) {

Reference Number: 327364-001
### Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgatherdpd</td>
<td>Gather float64 vector $U_{f64}(mv_t)$ into float64 vector zmm1 using doubleword indices and k1 as completion mask.</td>
</tr>
</tbody>
</table>

Note the special mask behavior as only a subset of the active elements of write mask k1 are actually operated on (as denoted by function $SELECT_{SUBSET}$). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, at least one element (the least significant enabled mask bit) will be selected from the source mask. Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the gather/scatter sequence have been loaded/stored and hence, the write-mask bits all are zero).
A set of 8 memory locations pointed by base address `BASE_ADDR` and doubleword index vector `VINDEX` with scale `SCALE` are converted to a float64 vector. The result is written into float64 vector `zmm1`.

Note the special mask behavior as only a subset of the active elements of write mask `k1` are actually operated on (as denoted by function `SELECT_SUBSET`). There are only two guarantees about the function: (a) the destination mask is a subset of the source mask (identity is included), and (b) on a given invocation of the instruction, at least one element (the least significant enabled mask bit) will be selected from the source mask.

Programmers should always enforce the execution of a gather/scatter instruction to be re-executed (via a loop) until the full completion of the sequence (i.e. all elements of the gather/scatter sequence have been loaded/stored and hence, the write-mask bits all are zero).
GATHER-LOADING ONLY YIELDS MODERATE LOOKUP IMPROVEMENT…

![Graph showing time per access in nanoseconds for different sizes of lookup table. The graph compares scalar, vectorized, and ratio performance.](image-url)
...SAME FOR PROJECTIONS

The graph shows the relationship between the stride in bytes and the time per access in nanoseconds. The graph compares two methods: Scalar and Vectorized, along with their ratio. The x-axis represents the stride in bytes (4, 8, 16, 32, 64, 128, 256, 512), while the y-axis represents the time per access in nanoseconds (0.03, 0.06, 0.13, 0.25, 0.50, 1.00, 2.00, 4.00). The Scalar method is represented by green squares, the Vectorized method by blue squares, and the ratio by red squares.
PREFETCHING
The PHI Prefetcher seems overly aggressive.

The graph shows the performance overhead of the PHI Prefetcher compared to bypassing it. The x-axis represents the stride in bytes, ranging from 4 to 4K. The y-axis represents the overhead ratio, ranging from 0.03 to 2.00.

The green dots represent the performance with the PHI Prefetcher enabled, while the blue dots represent the performance with the PHI Prefetcher bypassed. The graph indicates that the PHI Prefetcher is more aggressive than expected, as shown by the increased overhead compared to the expected behavior represented by the green line.
ONLY WHEN FACTORING IN TRANSFER OVERHEAD IS THE NOMINAL PHI BANDWIDTH ACHIEVED
TAKEAWAY

• Phi is en-par with mid-level GPUs compute-intensive applications

• Data-intensive performance is weird, though:
  • Prefetcher seems overly aggressive
  • Gather implementation seems half-baked: to few cache ports?
THANK YOU