Lecture 16: Parallel Architecture -- Data Level Parallelism

Concurrent and Multicore Programming

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

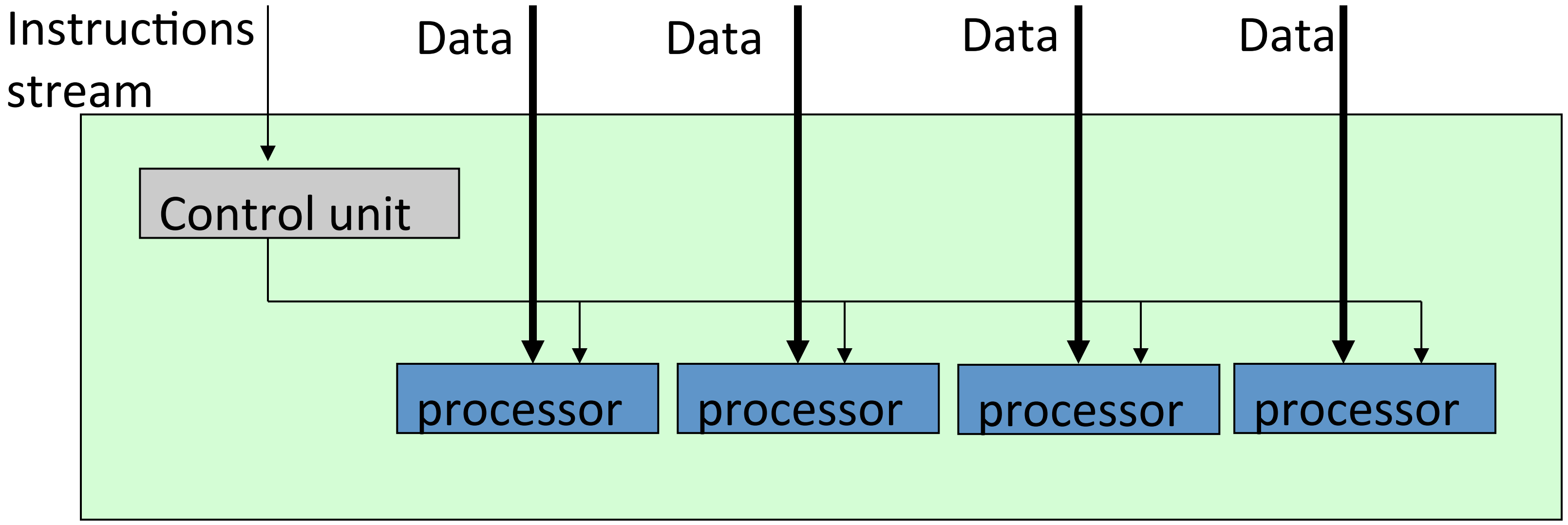
Classification of Parallel Architectures

Flynn's Taxonomy

- SISD: Single instruction single data
  - Classical von Neumann architecture
- SIMD: Single instruction multiple data
  - Vector, GPU, etc
- MISD: Multiple instructions single data
  - Non existent, just listed for completeness
- MIMD: Multiple instructions multiple data
  - Most common and general parallel machine

Single Instruction Multiple Data

- Also known as Array-processors
- A single instruction stream is broadcasted to multiple processors, each having its own data stream
  - Still used in some graphics cards today



SIMD Instructions

- Originally developed for Multimedia applications
- Same operation executed for multiple data items
- Uses a fixed length register and partitions the carry chain to allow utilizing the same functional unit for multiple operations
  - E.g. a 64 bit adder can be utilized for two 32-bit add operations simultaneously
- Instructions originally not intended to be used by compiler, but just for handcrafting specific operations in device drivers
- All elements in a register have to be on the same memory page to avoid page faults within the instruction

SIMD Instructions

- MMX (Mult-Media Extension) - 1996
  - Existing 64 bit floating point register could be used for eight 8-bit operations or four 16-bit operations
- SSE (Streaming SIMD Extension) – 1999
  - Successor to MMX instructions
  - Separate 128-bit registers added for sixteen 8-bit, eight 16-bit, or four 32-bit operations
- SSE2 – 2001, SSE3 – 2004, SSE4 - 2007
  - Added support for double precision operations
- AVX (Advanced Vector Extensions) - 2010
  - 256-bit registers added

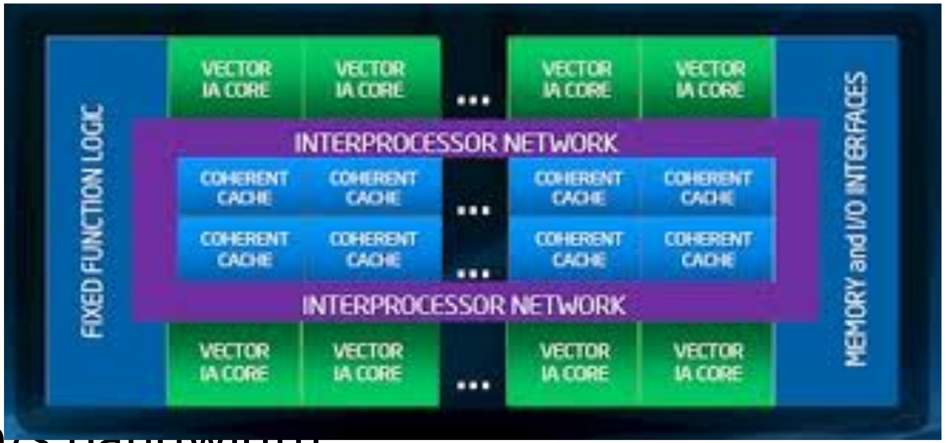
AVX Instructions

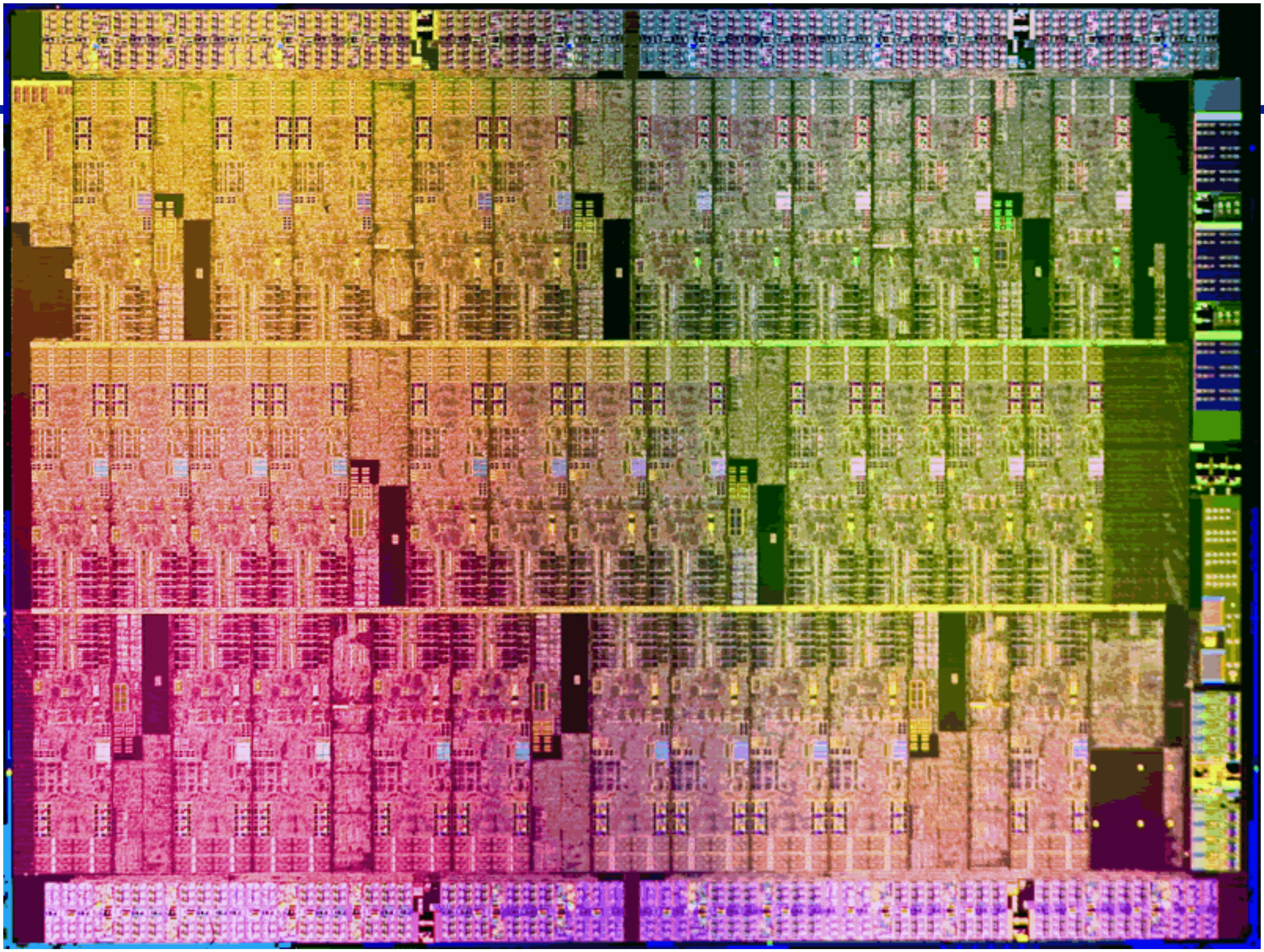
AVX Instruction	Description
`VADDPD`	Add four packed double-precision operands
`VSUBPD`	Subtract four packed double-precision operands
`VMULPD`	Multiply four packed double-precision operands
`VDIVPD`	Divide four packed double-precision operands
`VFMADDPD`	Multiply and add four packed double-precision operands
`VFMSUBPD`	Multiply and subtract four packed double-precision operands
`VCMPxx`	Compare four packed double-precision operands for `EQ, NEQ, LT, LTE, GT, GE`…
`VMOVAPD`	Move aligned four packed double-precision operands
`VBROADCASTSD`	Broadcast one double-precision operand to four locations in a 256-bit register

Intel Xeon Phi Processor

- First generation of Intel MIC (Many Integrated Cores) architecture

- 60 cores / 1.0 GHz
- 512-bit wide vector engine
- 32 Kb L1 I/D cache,
- 512 Kb L2 cache (per core)
- Up to 1 TFLOPS double-precision
- 8 Gb GDDR5 memory and 320 Gb/s bandwidth
- Standard PCIe x16 form factor





Multiple Instructions Multiple Data (I)

- Each processor has its own instruction stream and input data
- Very general case
  - every other scenario can be mapped to MIMD
- Further breakdown of MIMD usually based on the memory organization
  - Shared memory systems
  - Distributed memory systems

Vector Processors

- Vector processors abstract operations on vectors, e.g. replace the following loop

```
   for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
   }
```

  by

```
 a = b + c; ➡ ADDV.D V10, V8, V6
```

- Some languages offer high-level support for these operations (e.g. Fortran90 or newer)

Main concepts

- Advantages of vector instructions
  - A single instruction specifies a great deal of work
  - Since each loop iteration must not contain data dependence to other loop iterations
    - No need to check for data hazards between loop iterations
    - Only one check required between two vector instructions
    - Loop branches eliminated

Basic vector architecture

- A modern vector processor contains
  - Regular, pipelined scalar units
  - Regular scalar registers
  - Vector units – (inventors of pipelining!)
  - Vector register: can hold a fixed number of entries (e.g. 64)
  - Vector load-store units

Comparison MIPS code vs. vector code

Example: `Y=aX+Y` for 64 elements

```
   L.D    F0, a            /* load scalar a*/
   DADDIU R4, Rx, #512     /* last address */
L: L.D    F2, 0(Rx)        /* load X(i) */
   MUL.D  F2, F2, F0       /* calc. a times X(i)*/
   L.D    F4, 0(Ry)        /* load Y(i) */
   ADD.D  F4, F4, F2       /* aX(I) + Y(i) */
   S.D    F4, 0(Ry)        /* store Y(i) */
   DADDIU Rx, Rx, #8       /* increment X*/
   DADDIU Ry, Ry, #8       /* increment Y */
   DSUBU  R20, R4, Rx      /* compute bound */
   BNEZ   R20, L
```

Comparison MIPS code vs. vector code (II)

Example: `Y=aX+Y` for 64 elements

```
L.D     F0, a            /* load scalar a*/
LV      V1, 0(Rx)        /* load vector X */
MULVS.D   V2, V1, F0     /* vector scalar mult*/
LV      V3, 0(Ry)        /* load vector Y */
ADDV.D V4, V2, V3        /* vector add */
SV      V4, 0(Ry)        /* store vector Y */
```

Overhead

- Start-up overhead of a pipeline: how many cycles does it take to fill the pipeline before the first result is available?

Unit	Start-up
Load/store	12
Multiply	7
Add	6

Convoy	Starting time	First result	Last result
`LV`	1	12	12+n-1
`MULVS    LV`	12+n	12+n+11	23+n+n-1
`ADDV`	23+2n	23+2n+5	28+2n+n-1
`SV`	28+3n	28+3n+11	39+3n+n-1

Vector length control

- What happens if the length is not matching the length of the vector registers?
- A vector-length register (VLR) contains the number of elements used within a vector register
- *Strip mining*: split a large loop into loops less or equal the maximum vector length (MVL)

Vector length control (II)

```
low =0;
VL  = (n mod MVL);
for (j=0; j < n/MVL; j++ ) {
  for (i=low; i < low + VL; i++ ) {
       Y(i) = a * X(i) + Y(i);
  }
  low += VL;
  VL   = MVL;
}
```

Vector stride

- Memory on typically organized in multiple banks
  - Allow for independent management of different memory addresses
  - Memory bank time an order of magnitude larger than CPU clock cycle
- Example: assume 8 memory banks and 6 cycles of memory bank time to deliver a data item
  - Overlapping of multiple data requests by the hardware

Vector stride (II)

- What happens if the code does not access subsequent elements of the vector

```
   for (i=0; i<n; i+=2) {
    a[i] = b[i] + c[i];
   }
```

  – Vector load ‘compacts’ the data items in the vector register (gather)
    - No affect on the execution of the loop
    - You might however use only a subset of the memory banks -> longer load time
    - Worst case: stride is a multiple of the number of memory banks

Chaining

- Example:
```
MULV.D    V1, V2, V3
ADDV.D    V4, V1, V5
```
- Second instruction has a data dependence on the first instruction: two convoys required
- Once the element V1(i) is has been calculated, the second instruction could calculate V4(i)
  - no need to wait until all elements of V1 are available
  - could work similarly as forwarding in pipelining
  - Technique is called *chaining*

Chaining (II)

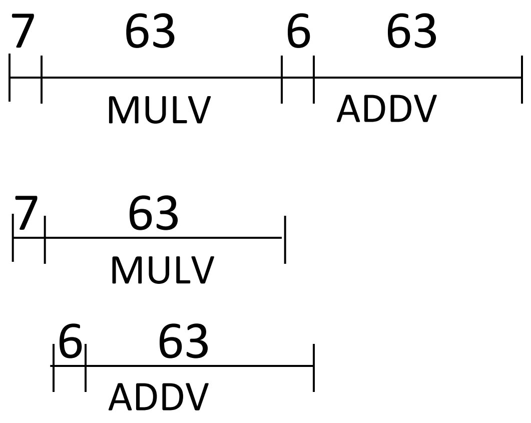
- Recent implementations use *flexible chaining*
  - Vector register file has to be accessible by multiple vector units simultaneously
- Chaining allows operations to proceed in parallel on separate elements of vectors
  - Operations can be scheduled in the same convoy
  - Reduces the number of chimes
  - Does not reduce the startup-overhead

Chaining (III)

- Example: chained and unchained version of the `ADDV.D` and `MULV.D` shown previously for 64 elements
  - Start-up latency for the FP MUL vector unit: 7cycles
  - Start-up latency for FP ADD vector unit: 6 cycles
- Unchained version:

  7 + 63 + 6 + 63 = 139 cycles
- Chained version:

  7 + 6 + 63 = 76 cycles



Conditional execution

- Consider the following loop

```
 for (i=0; i< N; i++ ) {
     if ( A(i) != 0 ) {
     A(i) = A(i) - B(i);
     }
 }
```

- Loop can usually not been vectorized because of the conditional statement
- Vector-mask control: boolean vector of length MLV to control whether an instruction is executed or not
  - Per element of the vector

Conditional execution (II)

```
LV        V1, Ra   /* load vector A into V1 */
LV        V2, Rb   /* load vector B into V2 */
L.D       F0, #0   /* set F0 to zero */
SNEVS.D   V1, F0   /* set VM(i)=1 if V1(i)!=F0 */
SUBV.D    V1, V1, V2  /* sub using vector mask*/
CVM             /* clear vector mask to 1 */
SV     V1, Ra   /* store V1 */
```

Support for sparse matrices

- Access of non-zero elements in a sparse matrix often described by

  `A(K(i)) = A(K(i)) + C (M(i))`

  - K(i) and M(i) describe which elements of A and C are non-zero
  - Number of non-zero elements have to match, location not necessarily
- Gather-operation: take an index vector and fetch the according elements using a base-address
  - Mapping from a non-contiguous to a contiguous representation
- Scatter-operation: inverse of the gather operation

Support for sparse matrices (II)

```
LV     Vk, Rk    /* load index vector K into V1 */
LVI    Va, (Ra+Vk) /* Load vector indexed A(K(i)) */
LV     Vm, Rm    /* load index vector M into V2 */
LVI    Vc, (Rc+Vm) /* Load vector indexed C(M(i)) */
ADDV.D Va, Va, Vc /* set VM(i)=1 if V1(i)!=F0 */
SVI    Va, (Ra+Vk) /* store vector indexed A(K(i)) */
```

- Note:
  - Compiler needs the explicit hint, that each element of K is pointing to a distinct element of A
  - Hardware alternative: a hash table keeping track of the address acquired
    - Start of a new vector iteration (convoy) as soon as an address appears the second time