INTRODUCTION TO RECONFIGURABLE COMPUTING

Volodymyr Kindratenko, NCSA
Craig Steffen, NCSA
Introduction to Reconfigurable Computing

• **Topics to be covered**
  • Introduction to the field of reconfigurable computing
  • A survey of basic concepts such as field-programmable gate arrays, dataflow, pipelining, etc.
  • Examples of modern reconfigurable computing systems
  • An overview of application development tools
  • Application development methodology
  • Examples of applications implemented on various reconfigurable computing platforms

• **Presenters**
  • Volodymyr Kindratenko and Craig Steffen, NCSA
Reconfigurable Computing (RC)

- Gerald Estrin's idea of “fixed plus variable structure computer”
  - reconfigurable hardware is tailored to perform a specific task as quickly as a dedicated piece of hardware
  - once the task is done, the hardware is adjusted to do other tasks
  - the main processor controls the behavior of the reconfigurable hardware

- Field Programmable Gate Array (FPGA) is the enabling technology

- FPGA vs. microprocessor
  - Microprocessor
    - fixed data path; software must be customized to make use of available busses and memory of the processor
  - FPGA
    - flexible data path; on-chip connectivity can be customized to fit the application

- IEEE Computer, March 2007
  High-Performance Reconfigurable Computers are parallel computing systems that contain multiple microprocessors and multiple FPGAs. In current settings, the design uses FPGAs as coprocessors that are deployed to execute the small portion of the application that takes most of the time—under the 10-90 rule, the 10 percent of code that takes 90 percent of the execution time.
FPGAs in HPC: Key Benefits

• By customizing hardware to match application computing and data needs, we can ‘build’ computers around applications rather than ‘wrap’ applications around computers

• Higher sustained performance can be attained
  • exploring inherent parallelism in algorithms
    • spatial parallelism, instruction level parallelism
  • matching computation with data flow

• Larger systems can be built with lower power budget
  • Example: 4 Altera Stratix III EP3SE260 chips consume an equivalent of Intel 3 GHz quad-core Xeon X5365 chip, thus giving a 5x performance advantage for the same WATTs of power

• FPGAs still have a significant room to grow, offering a potential for performance improvements over the microprocessors
Trends in FPGA Peak Performance

CPU: Moore’s Law - doubling in performance every 18 months

FPGA: performance is increasing by 4x every two years

# FPGAs vs. Multicore Processors

<table>
<thead>
<tr>
<th>Device</th>
<th>Theoretical Peak (GFLOPS) (*)</th>
<th>‘Practical’ Peak (GFLOPS)</th>
<th>Power (WATT)</th>
<th>GFLOPS/WATT (2)</th>
<th>Cost ($)</th>
<th>MFLOPS/$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD 2.8 GHz dual-core Opteron 8220</td>
<td>11.2</td>
<td>10.1(**)</td>
<td>95(†)</td>
<td>0.1</td>
<td>1,266(†)</td>
<td>8.0</td>
</tr>
<tr>
<td>Intel 3 GHz quad-core Xeon X5365</td>
<td>24</td>
<td>21.6(**)</td>
<td>120(†)</td>
<td>0.2</td>
<td>1,285(†)</td>
<td>16.8</td>
</tr>
<tr>
<td>Xilinx Virtex4 LX200</td>
<td>15.9(‡)(1) (@185 MHz)</td>
<td>9.1(***))</td>
<td>25(2)</td>
<td>0.4</td>
<td>10,589(‘’)</td>
<td>0.9</td>
</tr>
<tr>
<td>Xilinx Virtex5 LX330</td>
<td>28(‡)(1) (@237 MHz)</td>
<td>16(***))</td>
<td>30(2)</td>
<td>0.5</td>
<td>12,346(‘’)</td>
<td>1.3</td>
</tr>
<tr>
<td>Altera Stratix II EP2S180</td>
<td>25.2(‡)(2) (@303 MHz)</td>
<td>14.4(***))</td>
<td>25(2)</td>
<td>0.6</td>
<td>10,688(‘’’)</td>
<td>1.4</td>
</tr>
<tr>
<td>Altera Stratix III EP3SE260</td>
<td>50.7(‡)(2) (@363 MHz)</td>
<td>28.9(***))</td>
<td>30(2)</td>
<td>0.9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*) Add/multiply, double-precision floating-point (DP FP)
**) 90% of theoretical peak
***) 33% maximum frequency decrease and 15% logic remaining unused; (see ref 2 for details)
†) Implementing DP FP multipliers using both hard and soft multiplier cores
‡) Implementing DP FP multipliers with hard multiplier cores only
2) Martin Langhammer, *Double Precision Floating Point on FPGAs*, RSSI, July 17-20, 2007
‘) http://www.pricewatch.com/cpu/ (as of 09/07/2007)
‘’) http://www.em.avnet.com/ (as of 09/07/2007)
‘‘’) http://www.altera.com/buy/buy-index.html (as of 09/07/2007)
FPGAs in HPC: Key Challenges

• The programming model
  • Hardware design vs. software implementation methodologies
    • More than just a tool or language issue: hardware design requires a different way of thinking
    • Even with the best C-to-HDL compilers, software developers must be “system-aware” to extract any satisfactory performance

Software:
\[
float z = a*x + b*y;
\]

Hardware:
FPGAs in HPC: Key Challenges

- **The programming model**
  - Parallel programming is hard
    - Parallel programming with fine-grained parallelism is even harder
  - Loose coupling between microprocessor and FPGA
    - Applications are constrained by the I/O bandwidth and control transfer overhead
    - ‘Detached’ execution model is not suitable for many applications

```c
(binsA, a3, b3) = for(j in <0 .. NPOINTS>)
{
    (xj, yj, zj, a2, b2) = readpoint(a1, b1, j);
    float:53.11 dot = xi * xj + yi * yj + zi * zj;
    int:8 indx = findbin(dot, binb);
    binsB = foreach (bin in binsB by ind)
        if (ind == indx) bin + 1 else bin;
} (binsB, a2, b2);
```
FPGAs in HPC: Key Challenges

- **The programming model**
  - Using FPGAs to accelerate common libraries
    - Difficult to find libraries that are widely used in many applications
    - Typical library subroutines are too “fine-grained” to benefit from FPGA acceleration
  - A more productive approach is to port small, but computationally intensive application kernels
    - 10% of the code responsible for 90% of the compute time
    - The call frequency of the function is key here

Must have enough work to justify the configuration, data transfer, and control transfer overheads.
the “what can I fit in the FPGA” question involves a critical tradeoff

The kernel must be LARGE enough that execution time is dominated by computation time, not control transfer overhead

HOWEVER

The code/loop must be SMALL enough that the control logic (loop control, variable propagation and so on) is SMALL enough that there is still room left over for execution logic. (In other words, no implementing a full network stack on the FPGA).
FPGAs in HPC: Key Challenges

• Tools
  • Need for high-level languages capable of implementing both software and hardware
    • Existing tools, such as SRC Carte, are restricted to particular hardware platforms
  • Hardware synthesis and placing & routing compared to software compilation is time-consuming
  • Manual platform mapping is still required
    • SW/HW partitioning, SW/HW interface, data transfer, synchronization among FPGAs, use of memories, sequence of run-time reconfigurations,…
    • Code and data flow profiling tools maturity
  • Portability
    • Application portability at the source code level is difficult to maintain among HPRC products from the same hardware vendor (SRC-6/7 product line might be the only exception) and it is virtually non-existent between HPRC platforms from different vendors
FPGAs in HPC: Key Challenges

• **Lack of standards**
  - CPU-FPGA hardware interface varies from vendor to vendor
    • PCI, HyperTransport, DIMM, FrontSide Bus, …
  - On-board memory type and layout vary from product to product
    • Memory type, number of memory banks, memory bus width, memory bandwidth, access patterns, …
  - FPGA programming interface
    • Each vendor provides its own API to interface with the FPGA co-processor
  - High Level Languages that target FPGAs

• **Technology maturity and cost**
  - FPGAs are viewed as co-processors
  - Systems without a CPU are not even considered (exception: Stanford BEE system, which is designed for simulating multi-processor systems)
  - Systems with large number of FPGAs and small number of CPUs are not considered either (exception: Maxwell machine, but good luck using it!)
  - FPGAs are expensive
FPGAs are small clusters of “low-level” logic, e.g.
- flip-flops
- lookup tables (LUTs)
- and connection grids
that can be reconfigured to implement “higher-level” operations

“Bitstream” is a complete configuration for the chip
Example: Xilinx Virtex 2 FPGAs

- **Virtex-II XC2V6000**
  - 33,792 slices
    - 67,584 4-input LUTs
    - 67,584 flip flops
  - 144 18x18 integer multipliers
  - 144 Block RAMs (2,592 Kbits total)
  - 1,104 User I/O

- **Virtex 2 Pro 2VP100**
  - 44,096 slices
    - 88,192 4-input LUTs
    - 88,192 flip flops
  - 444 18x18 integer multipliers
  - 444 Block RAMs (7,992 Kbits total)
  - 1,164 User I/O
  - 20 RocketIO Transceivers
  - 2 PPC405s
FPGA Technology Trends

Max Frequency (MHz)

Configurable Logic Blocks (CLBs)

Block RAM (Kbytes)

Dedicated multipliers

Configurable Logic Blocks (CLB) slice

- Main elements are
  - lookup tables &
  - flip-flops
- *Configurable* refers to the ability to load lookup tables with user-specified logic
Lookup tables (LUT)

- Lookup tables are primary elements for logic implementation
- Each LUT can implement any function of 4 inputs
Implementing Operations on FPGA

- **Example: adder**
  - Described by a HDL (VHDL or Verilog)
  - “Synthesized” to the “low-level” resources available on the chip

![Adder Diagram]
Dataflow Concept

• **Basic idea**
  - express computation with interconnected function units
    - Data flow graph (DFG)

• **Can be implemented in FPGA logic**

• **Each function unit has a latency**
  - If we provide inputs to the DFG, the results will be output n clock cycles later
  - Thus, new inputs can be taken every n clock cycles
Pipelining Concept

- **Basic idea**
  - Non-pipelined functional unit can take new inputs only after it is done processing previous inputs
  - The fully pipelined functional unit can take a new input and produce a new output on every clock cycle
- **DFG can be pipelined by adding delays**
Examples of Pipelined DFGs

for (i=0; i<n; i++)
    a[i]=b[i]*c[i]

if (a>b) c = a+b;
else c = a*b;

```
&a
&b
&c

i

+

1c

+

1c

+

1c

delay

3c

load

3c

load

3c

delay

2c

X

2c

delay

1c

delay

1c

selector

1c

store

3c
```
Traditional FPGA “Code” Design Cycle

1. Algorithm
   - HDL Model
     - synthesis
     - netlist
       - implementation (map, place & route)
     - bitstream
       - downloading and testing

- Functional simulation
- Post-synthesis simulation
- Timing simulation
Mapping

LUT 1

LUT 2
LCELL

LUT 3

LUT 4
CARRY

carry_in1

netlist

carry_in2

a1
b1
Placing

LUT 1  LUT 2

LUT 3  LUT 4

netlist

FPGA

CLB slices
Routing

LUT 1

LUT 2

LUT 3

LUT 4

FPGA

CLB slices

netlist

RSSI 2008 Reconfigurable Computing Tutorial
P&R Report Example

• Device Utilization Summary

Number of BUFOMUXs                  1 out of 16      6%
Number of External IOBs           815 out of 1104   73%
    Number of LOCed IOBs           815 out of 815   100%
Number of MULT18X18s               10 out of 144     6%
Number of SLICEs                 3286 out of 33792   9%

• Clock report

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Requested</th>
<th>Actual</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Levels</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TS_CLOCK = PERIOD TIMEGRP &quot;CLOCK&quot; 10 ns H</td>
<td>10.000ns</td>
<td>9.786ns</td>
<td>0</td>
</tr>
<tr>
<td>IGH 50%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
High-Level Language based FPGA Code Design

Algorithm

HLL Model

HLL to HDL compiler

Synthesis and implementation transparent to the software developer

Conventional software execution and debugging

bitstream

downloading and testing
RC System Basic Concept

- Microprocessor
- Reconfigurable processor

disk

memory

microprocessor

common memory

memory

FPGA

communication channel (PCI, DIM, HyperTransport, etc.)
HPRC Systems Taxonomy

• uniform node nonuniform systems (UNNSs)

\[ \mu P_1 \cdots \mu P_N \] \[ \mu P_1 \cdots \mu P_N \] \[ \text{IN and/or GSM} \]

• nonuniform node uniform systems (NNUSs)

\[ \mu P_1 \ \cdots \ \mu P_N \] \[ \text{IN and/or GSM} \]

SRC-7 (UNNS)

Existing Networks

Courtesy of SCR Computers
SRC-7 (MAP H) Reconfigurable Processor

 Courtesy of SCR Computers

4.2 GB/s

14.4 GB/s sustained payload (7.2 GB/s per pair)

MAP

Controller EP2S130

User Logic 1

55 Mgates EP2S180

4.8 GB/s

User Logic 2

55 Mgates EP2S180

19.2 GB/s (2.4 x8)

Eight Banks On-Board Memory (64 MB SRAM)

4.2 GB/s

14.4 GB/s

12 GB/s

GPIO

1 GB SDRAM

1 GB SDRAM

4.8 GB/s

14.4 GB/s

19.2 GB/s (2.4 x8)
Cray

- **XD1 (NNUS)**

- **XT5h (NNNS)**
Application Acceleration Co-Processor

HyperTransport

3.2 GB/s

3.2 GB/s

Neighbor Compute Module

Accelaration FPGA

QDR II SRAM

QDR II SRAM

Neighbor Compute Module

RapidArray

2 GB/s

2 GB/s

2 GB/s

2 GB/s

3.2 GB/s

3.2 GB/s

3.2 GB/s

Courtesy of Cray
SGI Altix 350 with RC100 Blade (UNNS)

- Dual-Itanium 2 motherboard: 1.4 GHz, 4 GB memory
- NUMALink 4
- 2 microprocessors
- Memory

Diagram:
- TIO ASIC
- PROM
- Loader FPGA
- Algorithm FPGA 1 (Virtex 4 LX200)
- Algorithm FPGA 2 (Virtex 4 LX200)
- QDR SRAM
- 3.2 GB/s each direction
- 1.6 GB/s each direction

Dual-blade chassis:
- RC100 blade 1
- RC100 blade 2

Courtesy of SGI
XtremeData XD1000

Dual Opteron motherboard

JTAG/I2C ports

I2C temp/volt monitor

FLASH

CPLD

ZBT SRAM

FPGA

1.6GB/s

1.6GB/s

800 MB/s

800 MB/s

5.4 GB/s

6.4 GB/s

DDR333 or DDR400

DDR333 or DDR400

Opteron CPU 0

South Bridge

Courtesy of XtremeData
And many more...
FPGA Software stack

- **FPGA (internal) design**
  - User+HDL OR
  - HLL compiler
  - Must know about FPGA interfaces and external memory

- **FPGA interface control (in some systems this layer is transparent)**
  - Configures FPGA from CPU
  - Transfers data in and out of FPGA
  - Exerts software control of FPGA functioning
  - Tightly coupled to system and accelerator hardware design

- **User software interface**
  - User program controls FPGA according to access granted in above layer
  - Can be function call, or macros
  - Coupled to FPGA interface control
Software Stack Illustrated

- Disk
- Microprocessor
- Memory
- Common Memory
- User Software Interface
- Communication Channel (PCI, DIM, HyperTransport, etc.)
- FPGA Design
- FPGA Interface Control
SRC Carte

- Carte encompasses all three software layers
- FPGA compiler is C-to-FPGA translator
  - Includes provisions to include hardware-tuned macros
- Interface layer is transparent to the user
  - Carte knows about the hardware configuration of every device
- DMA transfers driven from FPGA side
- Software operates FPGA as a C function call
  - DMA blocks are passed as parameters
Mitrion-C

- Mitrion-C compiler configures the Mitrion Processor out of pre-defined hardware-tuned elements
  - Mitrion compiler tool encompasses hardware interface
- Software interface is explicit function calls
- Data movement is through CPU-side DMA transfers
DIME tools (Nallatech)

- Dime-C is a C-to-FPGA compiler
- DimeTalk is a hardware-oriented design tool. It designs a DIMETalk network for network-type communications between components
- Software interface is explicit DIMETalk API calls to control DMA and FPGA functioning
AutoPilot and Impulse-C

You will see later today
And many more…

- DSPL Logic
- CHiMPS
- Trident
- ROCCC
- ...
RC Software Development Cycle

- Algorithm specifications
- HW/SW partitioning
- FPGA platform description
- SW implementation, test & verification
- FPGA design
- HW implementation, test & verification
- microprocessor platform description
- microprocessor code
How to get started with porting an application to an RC system

- **Code profiling:** identify if the application has a compute kernel
  - 10-90 rule: 10% of the code responsible for 90% of the compute time
  - At least $O(N^2)$ complexity
  - Called infrequently

- **Understand computational budget of the kernel**
  - How many operations, what sort of operations
  - Will it fit onto FPGA(s)?

- **Data flow profiling:** understand kernel’s data requirements
  - Data types used, amount of input and output data, intermediate data storage needs
  - Will (and how) it fit into FPGA-accessible memory?

- **Run pen-and-paper calculations to get an estimate of the compute time on the FPGA**
  - How much time will it take to transfer data in and out?
  - How long will it take to run the calculations?

- **Port the kernel ‘as is’ with as little as possible changes made to it with the goals**
  - to verify correctness of the implementation
  - to understand space requirements and data storage needs
  - to identify possible performance bottlenecks

- **Optimize the initial implementation as long as system resources permit it**
Example: MATPHOT
(Stellar Photometry and Astrometry with Discrete Point Spread Functions)

• The MATPHOT algorithm achieves accurate and precise stellar photometry and astrometry of undersampled CCD observations by using supersampled discrete PSFs that are sampled 2, 3, or more times more finely than the observational data

• The MATPHOT algorithm shifts discrete PSFs within an observational model using a 21-pixel-wide damped sinc function
  • Image on the left is the simulated CCD image data with photon noise and electronic readout noise
  • Image on the right is MATPHOT's best model of that observation which is free of noise
  • The model is a degraded version of the supersampled Point Spread Function

• Image convolution using a separable 21-coefficient kernel is the computational core of the algorithm:

\[
 f_{\text{shifted}}(x_0) \equiv \sum_{i=-10}^{10} f(x_i) \frac{\sin \left( \frac{\pi (x_i - x_0)}{\pi (x_i - x_0)} \right)}{\pi (x_i - x_0)} \exp \left( - \left[ \frac{x_i - x_0}{3.25} \right]^2 \right)
\]

Source: Dr. Kenneth Mighell, National Optical Astronomy Observatory
Original C Implementation

• 2D convolution subroutine (90%)

/* shift DELTAX pixels in the X direction */
for (iy = 0; iy < image_in->sn; ++iy)
{
    for (ix = 0; ix < image_in->sm; ++ix)
        iAx[ix] = image_in->img[iy*image_in->sm+ix];

    sshift(iAx, image_in->sm, zeroF, oAx, sinc_x);
}

/* shift DELTAY pixels in the Y direction */
for (ix = 0; ix < image_in->sm; ++ix)
{
    for (iy = 0; iy < image_in->sn; ++iy)
        iAy[iy] = image_in->img[iy*image_in->sm+ix];

    sshift(iAy, image_in->sn, zeroF, oAy, sinc_y);
}

• 1D convolution subroutine (40%)

void sshift (float *x, int n, float hole, float *xp, float *sinc)
{
    /* convolve the input data with the sinc array */
    for (int point = 0; point < n; point++)
    {
        xp[point] = 0.0f;
        for (int lobe = 0; lobe < 21; lobe++)
        {
            int npix = point - (lobe - 10);
            if ( (npix >= 0) && (npix < n) )
            {
                xp[point] += sinc[lobe] * x[npix];
            }
            else
            {
                xp[point] += sinc[lobe] * hole;
            }
        }
    }
}

• Taken from MATPHOT software developed by Dr. Kenneth Mighell from the National Optical Astronomy Observatory

• Image occupies a continuous memory segment
From model to FPGA implementation

\[ a[m, n] \otimes h[k, l] = \sum_{i=0}^{k-1} \left\{ \sum_{j=0}^{l-1} a[m+i, n+j]h_{row}[j] \right\} h_{col}[i] \]

primary chip

- transfer in 2 sets of coefficients
- load ‘x’ set of coefficients into on-chip registers
- transfer image data to OBM banks
- compute convolution for each row, two pixels at a time
- transfer image data out to the host

secondary chip

- OBM E, F
- load ‘y’ set of coefficients into on-chip registers
- OBM A-C
- OBM D-F
- compute convolution for each column, two pixels at a time
- OBM A-C

SRC-6 MAP Series C reconfigurable processor
Hardware implementation

- Primary FPGA

  INNER LOOP pipelining
  loop on line 73:
    clocks per iteration: 1
    pipeline depth: 110

  PLACE AND ROUTE SUMMARY
  Number of Slice Flip Flops:
    44,802 out of 67,584 66%
  Number of 4 input LUTs:
    36,312 out of 67,584 53%
  Number of occupied Slices:
    33,790 out of 33,792 99%
  Number of MULT18X18s:
    132 out of 144 91%

  freq = 100.0 MHz

- Secondary FPGA

  INNER LOOP pipelining
  loop on line 69:
    clocks per iteration: 1
    pipeline depth: 118

  PLACE AND ROUTE SUMMARY
  Number of Slice Flip Flops:
    44,024 out of 67,584 65%
  Number of 4 input LUTs:
    36,641 out of 67,584 54%
  Number of occupied Slices:
    33,790 out of 33,792 99%
  Number of MULT18X18s:
    138 out of 144 95%

  freq = 100.1 MHz
/* compute sinc functions */
init_sinc_array_MAP(dx, sinc_x);
init_sinc_array_MAP(dy, sinc_y);

intp_filter ((int64_t *)image_in->img,
(int64_t *)image_out->img,
(int)image_in->sm, (int)image_in->sn,
(int64_t *)sinc_x, (int64_t *)sinc_y, zeroF,
&tm, mapnum);
Typical FPGA kernel implementation scenario

• *Application kernel* is a loop (or a group of loops) that operate(s) on a given data set

• **Porting a kernel to an FPGA usually requires**
  • Re-shaping data to fit FPGA memory requirements
  • Transferring data to FPGA memory
  • Implementing loop body on the FPGA
  • Transferring data back to the system memory
  • Copying results to the appropriate application memory

• **An efficient loop body implementation is the key to a successful FPGA kernel implementation**
  • Attention must be paid to the data issue as well
Data interdependence REQUIRED for good Pipeline performance

iteration 6 initial inputs available at “*” inputs

iteration 3,4,5 “*” computations in progress

iteration 2 results have been computed in “*”s

iteration 1 result has been computed in “+”

iteration 0 result being stored

fetching operands for iteration 7
Loop Performance

• **Example**
  
  ```
  for (k = 0; k < N; k++) {
  }
  ```

• **Pipeline depth**
  
  • clocks per iteration: 1
  
  • pipeline depth: 17

• **Execution time**
  
  • \((N + \text{pipeline depth})\)
Nested Loops Performance

**Example**

```c
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        k=i+j*N;
    }
}
```

**Pipeline depth**
- clocks per iteration: 1
- pipeline depth: 22

**Execution time**
- \((N + \text{pipeline depth}) \times M\)
Nested Loops Fusion

• **Before**
  
  ```
  for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
      k = i + j * N;
    }
  }
  ```

• **Pipeline depth**
  - clocks per iteration: 1
  - pipeline depth: 22

• **Execution time**
  - \((N + \text{pipeline depth}) \times M\)

• **After**
  
  ```
  for (l = 0; l < M * N; l++) {
    i = l / N;
    j = l % N;
    k = i + j * N;
  }
  ```

• **Pipeline depth**
  - clocks per iteration: 1
  - pipeline depth: 56

• **Execution time**
  - \(M \times N + \text{pipeline depth}\)
Loop slowdown

- Compiler attempts to pipeline the innermost loop
  - One clock cycle per loop iteration
  - Total loop execution time is $N+L$
    - $N$ is number of loop iterations, $L$ is loop latency

- Certain loop characteristics may prohibit loop pipelining
  - Multiple clock cycles per single loop iteration
  - Total loop execution time is $N \times D + L$
    - $D$ is loop “firing” rate

- Loop slowdown amounts to slower code execution and should be eliminated
Loop slowdown causes

• **Loop-carried scalar dependencies**
  - Scalar value generated in one loop iteration is referenced in the subsequent iteration

• **Loop-carried memory dependencies**
  - Writing to a memory location in one iteration which might be read in the subsequent iteration

• **Multiple access to the same memory bank**
  - Single-ported nature of OBM (and BRAM)
Example: Loop carried scalar dependence

```c
a=0;
for (i=0; i<N ;i++) {
    a = a + (i * (i + 2 ));
}
```

- The per-iteration loop delay cannot be less than the latency of the time to calculate the increment to “a”
- Some compilers have structures that enable this type of computation.
Example: Loop-carried memory dependence

```c
for (i=0; i<N; i++) {
}
```

- The contents of C[i] are not known at compile time. It is possible that there will be a read/write contention if the same element of A[] is read or written the read/write window. Compilers may, by default, increase the loop latency to make the behavior deterministic.

- Some compilers may have a switch that allows the programmer to turn off such considerations. This causes the loop to run fully pipelined, but if there is contention, the behavior will be undefined.
Example: Multiple accesses to memory bank

```c
for (k = 0; k < N; k++) {
}
```

- Arrays are often instantiated in Block RAM on the FPGA
- You cannot in general reference more than one value per array per clock
- Although there is no read-write contention, this loop, without modification, would have a delay of 3 clocks to accommodate the multiple memory reads.
Overlapping data movement and computations

- Conventional data flow
- Data flow using streams
Overlapping data movement and computations

- mapws 178 > ./ex04 10000
  - 11069 clocks (input DMA)
  - 20060 clocks (2 compute loops)
  - 10355 clocks (output DMA)
  - 41484 clocks (total)

- mapws 179 > ./ex04 10001
  - 11129 clocks (input DMA)
  - 20062 clocks (2 compute loops)
  - 10475 clocks (output DMA)
  - 41666 clocks (total)

- Note that when the element count goes up by 1, the compute time goes up by 2
Overlapping data movement and computations

- **mapws 175 > ./ex05 10000**
  - 11063 clocks (input DMA)
  - 10038 clocks (2 compute loops)
  - 10433 clocks (output DMA)
  - ---------
  - 31534 clocks (total)

- **mapws 176 > ./ex05 10001**
  - 11189 clocks (input DMA)
  - 10039 clocks (2 compute loops)
  - 10487 clocks (output DMA)
  - ---------
  - 31715 clocks (total)

- Note that when the element count goes up by 1, compute time only goes up by 1
Overlapping data movement and computations

- mapws 181 > ./ex06 10000
  - 11070 clocks (streaming DMA in and 2 compute loops)
  - 10379 clocks (output DMA)
  - ---------
  - 21449 clocks (total)

- mapws 182 > ./ex06 10001
  - 11190 clocks (streaming DMA in and 2 compute loops)
  - 10445 clocks (output DMA)
  - ---------
  - 21635 clocks (total)

- Note that compute is “free”!!!
Overlapping data movement and computations

- mapws 274 > ./ex07 10000
  - 11568 clocks (streaming dma in, 2 compute loops, streaming dma out)
- mapws 275 > ./ex07 10000
  - 11566 clocks (streaming dma in, 2 compute loops, streaming dma out)
- mapws 283 > ./ex07 64000
  - 65712 clocks (streaming dma in, 2 compute loops, streaming dma out)
- mapws 284 > ./ex07 64000
  - 68520 clocks (streaming dma in, 2 compute loops, streaming dma out)

- Note: get everything done for the cost of a "single" data movement
Maximize code use

- Each block of text lays down permanent execution hardware
- It is better to execute something when not needed and discard the result than to control execution with an “if” statement (goes with maximum collapsing of loops)
Code re-use example: $c=A \cdot B$ (dot product)

```c
float c = A[0] * B[0];
for (int i = 1; i < N; i++) {
    c += A[i] * B[i];
}
```

- // this uses TWO DIFFERENT potentially large execution units

```c
float c = 0.0;
for (int i = 0; i < N; i++) {
    c += A[i] * B[i];
}
```
Homogenize loops to help compiler

for (i...) {
    if (inputs are valid) {
        // do computations
        A[k++] = result;
    }
}

• Instead:
for (i...) {
    // do computations
    A[k] = result;
    if (inputs are valid) k++;
}
NAMD

\[ F(x_i) := \sum_{i \neq j=1}^{N} f(x_i, x_j) \]

\[ F(x_i) = m_i \frac{d^2 x_i}{dt^2} \]

Execution time \(~3.07\) seconds

\(~0.15\) seconds due to data DMA in/out and (measured on MAP)

\(~0.84\) seconds due to MAP function call overhead

\(~2.08\) seconds due to actual calculations (measured on MAP)

3x speedup
TPACF

$$\omega(\theta) = \frac{1}{n_D} \cdot DD(\theta) - \frac{2}{n_D n_R} \sum DR_i(\theta) + 1$$

$$= \frac{1}{n_R} \sum RR_i(\theta)$$

![Diagram with points and lines representing data points and relationships.](image)

![Graph showing speedup, execution time, and dataset size.](image)
TPACF: simple and repetitive wins

• **Function to determine bin value**
  
  ```
  dot = xi * data2[j].x + yi * data2[j].y + zi * data2[j].z;
  arcmin = r2A * acos(dot);
  indx = 1 + floor(bins_per_dec * (log10(arcmin) + binoffset));
  ```

• **How implemented in FPGA**
  
  ```
  dot = xi * data2[j].x + yi * data2[j].y + zi * data2[j].z;
  
  if (dot >= binb00) indx = 0;
  else if (dot >= binb01) indx = 1;
  else if (dot >= binb02) indx = 2;
  
  ...
  
  else indx = 32;
  ```

• **Multiple comparators are “cheaper” in hardware than one large function definition**
Two-electron repulsion integrals

\[
(\mu \nu | \lambda \sigma) = \sum_{p=1}^{N_{\mu}} \sum_{q=1}^{N_{\nu}} \sum_{r=1}^{N_{\lambda}} \sum_{s=1}^{N_{\sigma}} d_{\mu p} d_{\nu q} d_{\lambda r} d_{\sigma s} [pq|rs]
\]

\[
[s_1 s_2 | s_3 s_4] = \frac{\pi^3}{AB \sqrt{A + B}} K_{12}(\mathbf{R}_{12}) K_{34}(\mathbf{R}_{34}) F_0 \left( \frac{AB}{A + B} [\mathbf{R}_P - \mathbf{R}_Q]^2 \right)
\]

\[
\begin{align*}
A &= \alpha_1 + \alpha_2 \\
B &= \alpha_3 + \alpha_4 \\
F_0(t) &= \frac{erf(\sqrt{t})}{\sqrt{t}}, \\
\mathbf{R}_{kl} &= \mathbf{R}_k - \mathbf{R}_l \\
\mathbf{R}_P &= \frac{A}{\alpha_1 \mathbf{R}_1 + \alpha_2 \mathbf{R}_2} \\
\mathbf{R}_Q &= \frac{B}{\alpha_3 \mathbf{R}_3 + \alpha_4 \mathbf{R}_4}, \\
K_{ij}(\mathbf{R}_{ij}) &= \exp \left( -\frac{\alpha_i \alpha_j}{\alpha_i + \alpha_j} [\mathbf{R}_i - \mathbf{R}_j]^2 \right)
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>Model 1</th>
<th>Model 2</th>
</tr>
</thead>
<tbody>
<tr>
<td># of atoms</td>
<td>30</td>
<td>64</td>
</tr>
<tr>
<td>Basis set</td>
<td>6-311G</td>
<td>STO-6G</td>
</tr>
<tr>
<td># of integrals</td>
<td>528,569,315</td>
<td>2,861,464,320</td>
</tr>
<tr>
<td># of reduction elements</td>
<td>3,146,010</td>
<td>2,207,920</td>
</tr>
<tr>
<td>SRC-6 host (sec)</td>
<td>70.55</td>
<td>518.90</td>
</tr>
<tr>
<td>SRC-6 MAP E (sec)</td>
<td>25.42</td>
<td>42.85</td>
</tr>
<tr>
<td>Speedup</td>
<td>2.8x</td>
<td>12.1x</td>
</tr>
</tbody>
</table>