Accelerator Design, Tradeoffs, and Benchmarking

Vivado HLS
MachSuite [ IISWC 2014 ]
Quantifying Acceleration [ ISLPED 2013 ]

Brandon Reagen,
Yakun Sophia Shao, Bob Adolf,
Gu-Yeon Wei, David Brooks
Harvard University
## Tutorial Outline

<table>
<thead>
<tr>
<th>Time</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 am – 9:00 am</td>
<td>Introduction</td>
</tr>
<tr>
<td>9:00 am – 10:00 am</td>
<td>Pre-RTL Simulation Framework: Aladdin</td>
</tr>
<tr>
<td>10:00 am – 10:30 am</td>
<td>Break</td>
</tr>
<tr>
<td>10:30 am – 11:30 am</td>
<td>CAD &amp; Benchmarks: HLS &amp; MachSuite</td>
</tr>
<tr>
<td>11:30 am – 12:00 pm</td>
<td>Aladdin Validation and Case Studies</td>
</tr>
<tr>
<td>12:00 pm – 1:30 pm</td>
<td>Lunch</td>
</tr>
<tr>
<td>1:30 pm – 2:15 pm</td>
<td>Analytical Modeling: Lumos</td>
</tr>
<tr>
<td>2:15 pm – 2:45 pm</td>
<td>Workload Characterization Tool: WIICA</td>
</tr>
<tr>
<td>2:45 pm – 3:00 pm</td>
<td>Hands-on Exercise: Set up</td>
</tr>
<tr>
<td>3:00 pm – 3:30 pm</td>
<td>Break</td>
</tr>
<tr>
<td>3:30 pm – 5:00 pm</td>
<td>Hands-on Exercise</td>
</tr>
</tbody>
</table>
Accelerator Design Tools and Workloads

- Introduction to HLS
- Coding for HLS
- Practices for Performance
- MachSuite
- Design Space Exploration
What is an Accelerator?

- Scratchpad Memories
- General Purpose Cores
- Shared SRAMs
- Protocol
- Big Core
- Sm Core
What is an Accelerator?

*Fixed Function ASIC*

Optical Flow Accelerator
-- *Specialized* data path
-- FSM controlled
-- Single function

Orders of Magnitude
Energy Efficiency
CAD Tools

Accelerator Design Flow
Traditional Hardware Design

• Hand Code RTL
  – Understand problem
    • SORT
  – Consider various solutions
    • MERGE vs. RADIX

• Single Design Point
  – Power
  – Performance
  – Area
Traditional Hardware Design

• Pros
  – HDLs (Verilog) are **very** expressive
    • Designer can tune design perfectly
  – Engineers know and **trust** Verilog
    • Industry standard
Traditional Hardware Design

• Cons
  – RTL is time consuming
  – Single implementations take weeks, months, to years
  – Intuition driven
    • Make decisions up front
    • Miscalculations can be costly
Traditional Hardware Design

• Cons
  – RTL is time consuming
  – Single implementations take weeks, months, to years
  – Intuition driven
    • Make decisions up front
    • Miscalculations can be costly
Traditional Hardware Design

• Cons
  – RTL is time consuming
  – Single implementations take weeks, months, to years
  – Intuition driven
    • Make decisions up front
    • Miscalculations can be costly

*No high level approach to understand design space*
High Level Synthesis
Vivado HLS

• Xilinx’s HLS solution
  – C to Gates has been a pipe dream for decades
  – Vivado HLS paving the way

• Considers all aspects of Hardware Design
High Level Synthesis

• Generate RTL from HLL
  – Restrict and abstract expressivity of HDL
  – Trust the compiler
  – Like writing C++ rather than ASM
High-Level Synthesis
Scheduling & Binding

• Scheduling
  – Which clock cycle an operation will occur
    – Takes into account the control, dataflow and user directives

• Binding
  – Maps operations to available hardware
    – Takes into account component delays, user directives
High Level Synthesis
Vivado HLS

• Specify desired hardware

• Apply directives for performance

• Peripheral Support
High Level Synthesis

In this talk

• Focus on data path design
  – Achieving high performance
    • Loop Unrolling
    • Loop Pipelining
    • Functional Unit Selection and Allocation
    • Memory partitioning
High Level Synthesis
In this talk

• Focus on data path design
  – Achieving high performance
High Level Synthesis
In this talk

• Focus on data path design
  – Achieving high performance
Accelerator Design, Tradeoffs, and Benchmarking

Vivado HLS
MachSuite [ IISWC 2014 ]
Quantifying Acceleration [ ISLPED 2013 ]

Brandon Reagen,
Yakun Sophia Shao, Bob Adolf,
Gu-Yeon Wei, David Brooks
Harvard University
# Tutorial Outline

<table>
<thead>
<tr>
<th>Time</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 am – 9:00 am</td>
<td>Introduction</td>
</tr>
<tr>
<td>9:00 am – 10:00 am</td>
<td>Pre-RTL Simulation Framework: Aladdin</td>
</tr>
<tr>
<td>10:00 am – 10:30 am</td>
<td>Break</td>
</tr>
<tr>
<td><strong>10:30 am – 11:30 am</strong></td>
<td><strong>CAD &amp; Benchmarks: HLS &amp; MachSuite</strong></td>
</tr>
<tr>
<td>11:30 am – 12:00 pm</td>
<td>Aladdin Validation and Case Studies</td>
</tr>
<tr>
<td>12:00 pm – 1:30 pm</td>
<td>Lunch</td>
</tr>
<tr>
<td>1:30 pm – 2:15 pm</td>
<td>Analytical Modeling: Lumos</td>
</tr>
<tr>
<td>2:15 pm – 2:45 pm</td>
<td>Workload Characterization Tool: WIICA</td>
</tr>
<tr>
<td>2:45 pm – 3:00 pm</td>
<td>Hands-on Exercise: Set up</td>
</tr>
<tr>
<td>3:00 pm – 3:30 pm</td>
<td>Break</td>
</tr>
<tr>
<td>3:30 pm – 5:00 pm</td>
<td>Hands-on Exercise</td>
</tr>
</tbody>
</table>
Accelerator
Design Tools and Workloads

• Introduction to HLS
• Coding for HLS
• Practices for Performance
• MachSuite
• Design Space Exploration
Leveraging HLS
Loop Unrolling

• By default, loops are rolled
  – Each C loop iteration
    • Implemented in the same state
  – Each C loop iteration
    • Implemented with same resources

```c
void foo_top (...)
{
  ...
  Add: for (i=3;i>=0;i--){
    b = a[i] + b;
  ...
  }
}  
```
Loop Unrolling

• Unrolled Loops increase
  – Performance, Area, Power

• Lower energy

```c
void foo_top (...) {
    ...
    Add: for (i=3;i>=0;i--){
        b = a[i] + b;
        ...
    }
```
Loop Pipelining

**Without Pipelining**

**Loop_tag** : for( II = 1 ; II < 3 ; II++ ) {
  op_Read;
  op_Compute;
  op_Write;
}

- Throughput = 3 cycles
- Latency = 3 cycles
- Loop Latency = 6 cycles

**With Pipelining**

- Throughput = 1 cycle
- Latency = 3 cycles
- Loop Latency = 4 cycles

Performance ➤ Area ➤ Power ➤
Loop Pipelining
Initiation Intervals

• Iteration interval (II) is the time loop must wait for the next execution to begin

An II=1 cannot be implemented
  – Port cannot be read at the same time
  – Similar effect with other resource limitations
    • For example if functions or multipliers etc. are limited
Hardware Resource
Selection and Allocation

- **Allocation** directive limits different types
  - Operations
    - Functional Units
  - Functions
    - Maps multiple instantiations to same RTL
Hardware Resource Selection and Allocation

- **Allocation** directive limits different types
  - Operation
    - Functional Units
  - Functions
    - Maps multiple instantiations to same RTL

- Select operation’s **implementation**
  - “Tag” functional units

\[
a[i] = b[i] \times c[i]
\]

\[
\text{thisMult} = b[i] \times c[i];
a[i] = \text{thisMul};
\]
Hardware Resource Selection and Allocation

- **Allocation** directive limits different types
  - Operation
    - Functional Units
  - Functions
    - Maps multiple instantiations to same RTL

- Select operation’s **implementation**
  - Select core for `thisMult`

```plaintext
thisMult = b[i] * c[i];
```

<table>
<thead>
<tr>
<th>Core</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mul</td>
<td>Combinational mult</td>
</tr>
<tr>
<td>Mul3s</td>
<td>3-Stage pipelined mult</td>
</tr>
<tr>
<td>MulnS</td>
<td>HLS determine stages</td>
</tr>
</tbody>
</table>
Array Partitioning

- Partitioning breaks an array into smaller elements

- Multiple memories allow greater parallel access
High Level Synthesis
Vivado HLS

• Covered most impactful directives
  – Loop Unrolling
  – Loop Pipelining
  – Memory Partitioning
  – Resource selection and allocation

• Many more exist..
  – ~30 explicit directives with multiple dimensions
    – With the correct directives and code, designs can often compete with handwritten RTL
Coding For HLS

In conjunction with Primal Patel @ Xilinx
HLS Compatible Code

• Language Support
  – What you CAN do

• Unsupported Language features
  – What you CAN NOT do

• Building on Language standards
  – Adding expressivity in HLLs
  – Coding Suggestions..
Language Support

• Vivado HLS supports C, C++, and SystemC
  – Provided static definition at compile time
  – run time definitions are not synthesizable

• Data types
  – Float and double cores generated
  – Supports arbitrary precision types
  – User defined fixed point types

• Validation & Verification Environment
  – Complete bit-accurate validation of the C model
  – C-RTL co-simulation verification solution
Unsupported Features
Dynamic Memory Allocation

• Dynamic memory allocation
  – Requires construction of hardware at runtime
    • malloc, alloc, free are not synthesizable
  – Use persistent static variables and fixed-size arrays

```
long long x = malloc (sizeof(long long));
int* arr = malloc (64 * sizeof(int));
```

```
static long long x;
int array[64];
```
Unsupported Features
System Calls

• System calls
  – C system calls do not have hardware
    – printf(), getc(), time(), ...
  – Vivado HLS will ignore system calls
    – Use “__SYNTHESIS__”
    – Synthesis will ignore block
    – Automatically defined

```c
void foo (...) {
    #ifndef __SYNTHESIS__
        Code will be seen by simulation.
    But not synthesis.
    #endif
}
```

Only read this code if macro __SYNTHESIS__ is not set
Unsupported Features
General Pointer Casting

- Pointer reinterpretation
  - Can’t cast a pointer to a different type

```c
int main()
{
    struct {
        short first;
        short second;
    } pair;
    *((unsigned*)pair) = -1U;
    pair = -1U;
    pair.second = -1U;
    return 0;
}
```

Solution
- Pointer casting **allowed** between native integer types

```c
struct {
    short first;
    short second;
} pair;
pair.first = -1U;
pair.second = -1U;
```
Unsupported Features
Recursive Functions

• Avoid recursive functions
  – Not synthesizable in general
    • Code re-entrance indirectly uses dynamic memory allocation

unsigned foo (unsigned n) {
  if (n == 0 || n == 1) return 1;
  return (foo(n-2) + foo(n-1));
}
Language Support

• Lots of support
• Pointers bring conservativeness
  – Memory disambiguation
• Suggestion
  – Help the compiler for best results
  – Limit pointer use
  – Limit potential for false dependencies

Powerful tool –

but conservatism for correctness can ruin performance
Implementation and Co-optimizations
Importance Implementation

• Consider Simple Application Kernel
  – In place SCAN
Importance Implementation

- Consider Simple Application Kernel
  - In place SCAN

Data = [1, 10, 4, 5]
Importance Implementation

• Consider Simple Application Kernel
  – In place SCAN

Data = [1, 10, 4, 5]  Scan(data)
Importance Implementation

• Consider Simple Application Kernel
  – In place SCAN

Data = [1, 10, 4, 5]  \rightarrow \text{Scan(data)}  \rightarrow \text{Data = [1, 11, 15, 20]}
Importance Implementation

• Consider Simple Application Kernel
  – In place SCAN

\[
Data = [1, 10, 4, 5] \quad \rightarrow \quad \text{Scan(data)} \quad \rightarrow \quad Data = [1, 11, 15, 20]
\]

\[
Data[x] = \sum_{i=0}^{i=x} Data[i]
\]
Importance Implementation

• Consider Simple Application Kernel
  – In place SCAN

*different frequencies
Importance Implementation

- Consider Simple Application Kernel
  - Resource Contention
  - Cannot resolve RAW hazard
Importance Implementation

• Consider Simple Application Kernel
  – Increases II
  – Instantiates more HW
    • But compiler cannot make good use of it
Importance Implementation

• Consider Simple Application Kernel
  – Increases II
  – Instantiates more HW
    • But cannot make good use of it
Importance Implementation

• Consider Simple Application Kernel
  – Increases II
  – Instantiates more HW
    • But cannot make good use of it
Importance of co-design

• Instead of the naïve, serial SCAN

Can we try something smarter?

What about parallel SCAN?
Importance of co-design

• Parallel SCAN

DATA

Local Scan

DATA1  DATA2  DATA15  DATA16
Importance of co-design

- Instead of the naïve, serial SCAN
Importance of co-design

• Instead of the naïve, serial SCAN
Importance of co-design

- Exposing parallelism explicitly to HLS tool helps
- Must still tune parallel code
Importance of co-design

• Exposing parallelism explicitly to HLS tool helps
• Must still tune parallel code
  – Co-optimize data access patterns and Array partitioning

How much difference can such tuning make?
A Lot.
What Happened?

![Graph showing power consumption over execution time with two lines: Tuned C Code and Unoptimized C Code. The green line has a minimum at 1667 cycles with a power of 0.5 mW, and the yellow line has a peak at 4611 cycles with a power of 4.0 mW.](image-url)
What Happened?

- Same directives
  - Single port SRAMs
  - Arrays factor 4 partition, cyclic
  - Outer Loops pipelined
    - Automatically completely unrolls inner loop

Same Algorithm
Different Implementation
What Happened?

• “Unoptimized C Code”
  – Pipelining result:
    Target II: 1, Final II: 30

• “Optimized C Code”
  – Pipelining result:
    Target II: 1, Final II: 8
What Happened?
Unoptimized C Code

for i = 1 : Block
    for radixID : Radix
        bucket[i*Block+radixID ] +=
            bucket[i*Block+ radixID-1];
What Happened?
Optimized C Code

for radixID : Radix
    for i = 1 : Block
        bucket[i*Block + radixID ] +=
        bucket[i*Block + radixID-1];
Solution

MEMORY

SCAN Accelerator

MEMORY

SCAN Accelerator
Solution

MEMORY

SCAN Accelerator

MEMORY

SCAN Accelerator
Solution

MEMORY

SCAN Accelerator

MEMORY

SCAN Accelerator
MachSuite:
An Accelerator Benchmark Suite
[ IISWC 2014 ]
MachSuite

• Motivation
  – No commensurability across publications
  – Just saw how much implementation matters
  – It takes a lot of time to write benchmarks...
    
    Doesn’t make sense!
Literature Survey

25 Papers

88 Different Benchmarks

64 used only **ONCE**
### MachSuite

<table>
<thead>
<tr>
<th>Kernel/Algorithm</th>
<th>Description</th>
<th>Berkeley Dwarf [5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES/AES</td>
<td>AES encryption</td>
<td>Combinational logic</td>
</tr>
<tr>
<td>BACKPROP/BACKPROP</td>
<td>Neural network training</td>
<td>Unstructured grids</td>
</tr>
<tr>
<td>BFS/BULK</td>
<td>Breadth-first search</td>
<td>Graph traversal</td>
</tr>
<tr>
<td>BFS/QUEUE</td>
<td>Breadth-first search</td>
<td>Graph traversal</td>
</tr>
<tr>
<td>DES/DES</td>
<td>DES encryption</td>
<td>Combinational logic</td>
</tr>
<tr>
<td>FFT/SIMPLE</td>
<td>Fast Fourier transform</td>
<td>Spectral methods</td>
</tr>
<tr>
<td>FFT/TRANSPOSE</td>
<td>Fast Fourier transform</td>
<td>Spectral methods</td>
</tr>
<tr>
<td>GEMM/SIMPLE</td>
<td>Matrix multiplication</td>
<td>Dense linear algebra</td>
</tr>
<tr>
<td>GEMM/BLOCKED</td>
<td>Matrix multiplication</td>
<td>Dense linear algebra</td>
</tr>
<tr>
<td>KMP/KMP</td>
<td>String matching</td>
<td>Finite state machines</td>
</tr>
<tr>
<td>MD/KNN</td>
<td>Molecular dynamics</td>
<td>N-body methods</td>
</tr>
<tr>
<td>MD/GGRID</td>
<td>Molecular dynamics</td>
<td>N-body methods</td>
</tr>
<tr>
<td>NW/NW</td>
<td>DNA alignment</td>
<td>Dynamic programming</td>
</tr>
<tr>
<td>SHA/SHA</td>
<td>SHA1 hashing</td>
<td>Combinational logic</td>
</tr>
<tr>
<td>SORT/MERGE</td>
<td>Sorting</td>
<td>Map reduce</td>
</tr>
<tr>
<td>SORT/RADIX</td>
<td>Sorting</td>
<td>Map reduce</td>
</tr>
<tr>
<td>SPMV/CRS</td>
<td>Sparse matrix/vector multiplication</td>
<td>Sparse linear algebra</td>
</tr>
<tr>
<td>SPMV/ELLPACK</td>
<td>Sparse matrix/vector multiplication</td>
<td>Sparse linear algebra</td>
</tr>
<tr>
<td>STENCIL/SIMPLE</td>
<td>Stencil computation</td>
<td>Structured grids</td>
</tr>
<tr>
<td>STENCIL/STENCIL3D</td>
<td>Stencil computation</td>
<td>Structured grids</td>
</tr>
<tr>
<td>VITERBI/VITERBI</td>
<td>Hidden Markov model estimation</td>
<td>Graphical models</td>
</tr>
</tbody>
</table>
## MachSuite

<table>
<thead>
<tr>
<th>Kernel/Algorithm</th>
<th>Description</th>
<th>Berkeley Dwarf [5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES/AES</td>
<td>AES encryption</td>
<td>Combinational logic</td>
</tr>
<tr>
<td>BACKPROP/BACKPROP</td>
<td>Neural network training</td>
<td>Unstructured grids</td>
</tr>
<tr>
<td>BFS/BULK</td>
<td>Breadth-first search</td>
<td>Graph traversal</td>
</tr>
<tr>
<td>BFS/QUEUE</td>
<td>Breadth-first search</td>
<td>Graph traversal</td>
</tr>
<tr>
<td>DES/DES</td>
<td>DES encryption</td>
<td>Combinational logic</td>
</tr>
<tr>
<td>FFT/SIMPLE</td>
<td>Fast Fourier transform</td>
<td>Spectral methods</td>
</tr>
<tr>
<td>FFT/TRANSPOSE</td>
<td>Fast Fourier transform</td>
<td>Spectral methods</td>
</tr>
<tr>
<td>GEMM/SIMPLE</td>
<td>Matrix multiplication</td>
<td>Dense linear algebra</td>
</tr>
<tr>
<td>GEMM/BLOCKED</td>
<td>Matrix multiplication</td>
<td>Dense linear algebra</td>
</tr>
<tr>
<td>KMP/KMP</td>
<td>String matching</td>
<td>Finite state machines</td>
</tr>
<tr>
<td>MD/KNN</td>
<td>Molecular dynamics</td>
<td>N-body methods</td>
</tr>
<tr>
<td>MD/GRID</td>
<td>Molecular dynamics</td>
<td>N-body methods</td>
</tr>
<tr>
<td>NW/NW</td>
<td>DNA alignment</td>
<td>Dynamic programming</td>
</tr>
<tr>
<td>SHA/SHA</td>
<td>SHA1 hashing</td>
<td>Combinational logic</td>
</tr>
<tr>
<td>SORT/MERGE</td>
<td>Sorting</td>
<td>Map reduce</td>
</tr>
<tr>
<td>SORT/RADIX</td>
<td>Sorting</td>
<td>Map reduce</td>
</tr>
<tr>
<td>SPMV/CRS</td>
<td>Sparse matrix/vector multiplication</td>
<td>Sparse linear algebra</td>
</tr>
<tr>
<td>SPMV/ELLPACK</td>
<td>Sparse matrix/vector multiplication</td>
<td>Sparse linear algebra</td>
</tr>
<tr>
<td>STENCIL/SIMPLE</td>
<td>Stencil computation</td>
<td>Structured grids</td>
</tr>
<tr>
<td>STENCIL/STENCIL3D</td>
<td>Stencil computation</td>
<td>Structured grids</td>
</tr>
<tr>
<td>VITERBI/VITERBI</td>
<td>Hidden Markov model estimation</td>
<td>Graphical models</td>
</tr>
</tbody>
</table>
Why not use existing GPU Benchmark?

• GPUs and Accelerators
  different approaches

  – CUDA/OpenCL targets pre-defined hardware (GPU)

  – Accelerators make sense when GPUs do not

  – Accelerators need to account for more than TLP
    • FU selection/allocation
    • ILP
    • Arbitrary memory hierarchy
Usability and Characterization

• MachSuite prides itself on usability
  – *MakeFile generates binaries and RTL!*

• Scripts to sweep directives
  – All loops, FUs, Memories are “tagged”

• Paper provides rigorous characterization
  – Application Space coverage
  – Locality, parallelism, etc....
Try it Out!

• BETA version of MachSuite on you’re Aladdin USB drive

• Public .git repository and pre-print paper coming soon

Register online and we’ll send you a link!
Design Space Exploration [ ISLPED 2013 ]
What HLS Enables

• Quantitative approach to DSE
  – Not as dependent on intuition

• Feedback from compiler as to what to tune
  – As seen before, keep II low

• Leave nothing on the table
  – Entire Design Space
  – How good is 10x?
Quantifying Acceleration

ISLPED 2013

• Explore large design spaces

• Analyze Pareto Frontier designs
  – Provide insight on performance scaling

• Compare Accelerator designs to GP-Core
Accelerator Design Flow

*note here we also swept FU implementations and frequencies*
Application Kernels

Simple workloads, huge design space

TRIAD

STENCIL
Instruction/Area Breakdown

![Graph showing Instruction/Area Breakdown]

- **Relative Workload Overhead/Compute Fractions**
  - Y-axis: 0% to 100%
  - Categories: M0 Triad, M0 Scan, M0 Reduction, M0 Stencil, M0 GEMM
  - Overhead (Yellow) and Compute (Blue)

[Graph legend: Overhead, Compute]
Different Energy Scaling

**TRIAD**

**STENCIL**

[Diagrams showing energy breakdown for TRIAD and STENCIL]
Conclusions

• HLS can generate high performing Hardware
  – If you do it right...

• Very sensitive to implementation
  – MachSuite offers benchmarking standard!

• Despite specificity, Huge Space!

• But really... I only care about Pareto Points
  – Energy breakdown example
Conclusions

• With *Aladdin*, you get the Pareto Points 100x faster!

• Avoid *cumbersome* C code tuning
  – More powerful analysis, less restrictions
## MachSuite (backup)

<table>
<thead>
<tr>
<th>Kernel/Algorithm</th>
<th>Description</th>
<th>Berkeley Dwarf [5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES/AES</td>
<td>AES encryption</td>
<td>Combinational logic</td>
</tr>
<tr>
<td>BACKPROP/BACKPROP</td>
<td>Neural network training</td>
<td>Unstructured grids</td>
</tr>
<tr>
<td>BFS/BULK</td>
<td>Breadth-first search</td>
<td>Graph traversal</td>
</tr>
<tr>
<td>BFS/QUEUE</td>
<td>Breadth-first search</td>
<td>Graph traversal</td>
</tr>
<tr>
<td>DES/DES</td>
<td>DES encryption</td>
<td>Combinational logic</td>
</tr>
<tr>
<td>FFT/SIMPLE</td>
<td>Fast Fourier transform</td>
<td>Spectral methods</td>
</tr>
<tr>
<td>FFT/TRANSPOSE</td>
<td>Fast Fourier transform</td>
<td>Spectral methods</td>
</tr>
<tr>
<td>GEMM/SIMPLE</td>
<td>Matrix multiplication</td>
<td>Dense linear algebra</td>
</tr>
<tr>
<td>GEMM/BLOCKED</td>
<td>Matrix multiplication</td>
<td>Dense linear algebra</td>
</tr>
<tr>
<td>KMP/KMP</td>
<td>String matching</td>
<td>Finite state machines</td>
</tr>
<tr>
<td>MD/KNN</td>
<td>Molecular dynamics</td>
<td>N-body methods</td>
</tr>
<tr>
<td>MD/GRID</td>
<td>Molecular dynamics</td>
<td>N-body methods</td>
</tr>
<tr>
<td>NW/NW</td>
<td>DNA alignment</td>
<td>Dynamic programming</td>
</tr>
<tr>
<td>SHA/SHA</td>
<td>SHA1 hashing</td>
<td>Combinational logic</td>
</tr>
<tr>
<td>SORT/MERGE</td>
<td>Sorting</td>
<td>Map reduce</td>
</tr>
<tr>
<td>SORT/RADIX</td>
<td>Sorting</td>
<td>Map reduce</td>
</tr>
<tr>
<td>SPMV/CRS</td>
<td>Sparse matrix/vector multiplication</td>
<td>Sparse linear algebra</td>
</tr>
<tr>
<td>SPMV/ELLPACK</td>
<td>Sparse matrix/vector multiplication</td>
<td>Sparse linear algebra</td>
</tr>
<tr>
<td>STENCIL/SIMPLE</td>
<td>Stencil computation</td>
<td>Structured grids</td>
</tr>
<tr>
<td>STENCIL/STENCIL3D</td>
<td>Stencil computation</td>
<td>Structured grids</td>
</tr>
<tr>
<td>VITERBI/VITERBI</td>
<td>Hidden Markov model estimation</td>
<td>Graphical models</td>
</tr>
</tbody>
</table>
Power Model (backup)

- Standard Cell Libraries
  - FreePDK 45nm
  - TSMC 40nm

- Design Compiler

- Activity Files

- Micro Benchmarks

- Power
- Delay
- Area