DSP processors

Overview of architecture and SDKs

Disclaimer

- Presentation focuses on DSP *compilers*, not architectures (but …)
- All information is publicly available
- All opinions are my own, not of my employer

Plan of the talk

- Why DSPs exist
- History of DSPs
- Target applications and how they shape DSPs
- Traditional and modern DSP architectures
- DSP ecosystem
- DSP tools
- Compilers as critical part of DSP products
- How DSP compilers are special (or not)
- Wrap-up

What is a DSP

- Digital Signal Processors
- Appeared in 1970-s
- First computational accelerators
- Built to compete with analog ASIC signal processors (“programmable Signal Processors”)
- Pros: reprogrammable => lower cost, reduced TTM, field updates in case of bugs, changes in standards, etc.
- Notoriously hard to program

ASIC – Application Specific Integrated Circuit
TTM – time-to-market

DSP vs general-purpose processors (GPPU)

	GPPU	DSP
Frequency	2-3 GHz	~500 MHz
Power consumption	Very high (~100 W)	Very low (~1 W)
Out-of-order	Yes	No
Price	$1000+	~$50
Target applications	General-purpose	Loop-intensive, data-parallel, highly regular

Over 100x Perf/W/$ improvement

Drinking from the Firehose: How the Mill CPU Decodes 30+ Instructions per Cycle

GPPU control overhead

"While a simple arithmetic operation requires around 0.5–20 pJ, modern cores spend about 2000 pJ to schedule it." [2]

[1] https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)
[2] The Rise and Fall of Dark Silicon
[3] Understanding sources of inefficiency in general-purpose chips

DSP: applications (1)

- Real-time embedded systems
  - Need predictable performance
  - **Power consumption is very important**
  - Single program (size of developed code is *smaller*)
  - Consists of few functions which implement of some standard (or performance-critical parts of it)
  - Each function is a bunch of loops
- Data processing is one of the main functions
  - Number crunching loops over linear arrays
  - Data-parallel or reductions (convolutions, stencils)
  - IIR/FIR, dot products, Fourier transforms, etc.
  - **Most common operations: additions and multiplications (MACs)**
  - **Unlimited ILP**

DSP: applications (2)

- Predictable data ranges
  - Fixed point arithmetic
  - Saturation on overflow (instead of wrap-around)
- Economy of scale
  - Consequence of embeddedness
  - Same program runs on millions of devices
  - **Program optimization has very high ROI**
  - Compile time or developer convenience is a lesser issue
  - Binary compatibility not an issue (source/assembler compatibility often is!)

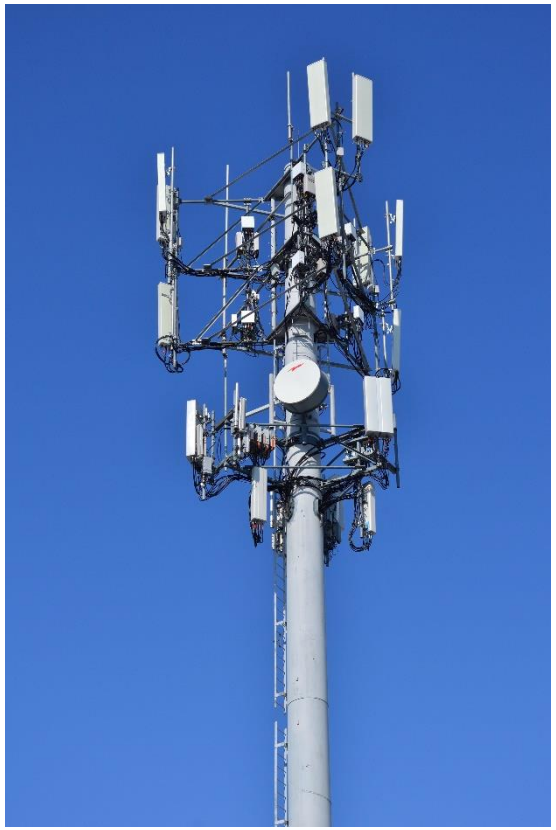
DSP success story: Speak&Spell

- Toy speech synthesizer
- Appeared in 1978
- Based on dedicated TI DSP
- Used Linear Predictive Coding for synthesis which allowed dramatic reduction of required ROM
  - Human vocal tract modelled as series of linear filters
  - Speech is a sequence of phonemes
  - Each phonem is determined by filter coefficients and excitation signal
- Audio processing still one of the main DSP markets (sound cards, codecs, etc.)

By FozzTexx at English Wikipedia

DSP success story: GSM

- First widely used digital mobile standard
- First deployed in 1991
- Originally planned to use ASICs but switched to DSPs as protocol complexity increased
- Example chip – TMS320C50



By Raysonho @ Open Grid Scheduler / Scalable Grid Engine

- Communications is one of the main DSP markets to date (modems, mobile base stations, etc.)

DSP success story: image processors

- Appeared in early 2000-s in digital cameras
- Various postprocessing (real-time denoising, sharpening, etc.) and encoding functions
- Example chip – TMS320C50
- Image/video processing and computer vision is one of the main DSP markets (cameras, surveillance systems in drones and automotive, etc.)

By Dave Dugdale from Superior, USA

5 Things Made Possible with DIGIC Image Processor

Gene's law

- Named after Gene Frantz from TI
- Gene's law:
  - Power dissipation of DSP cores halves every 1.5 years
- Or alternatively Koomey's law:
  - GFLOPS/W doubles every 1.5 years
- Consequence of Moore's law ...
  - Number of transistors doubles every 1.5 years
- ... and Dennard scaling:
  - Power density of chip remains constant as transistors get smaller
- Allowed DSPs to cover more computationally-intensive markets over years while still staying into 1W power limit
  - From sound synthesizers to video, AI, SDR
- No longer working :(

DSP: architecture

- “Architecture of DSP is molded by algorithms“:
  - Unlimited ILP (in loops)
  - High-school math computations (complex math, convolutions, Fourier series, etc.)
  - No need for binary compatibility
- Two key features:
  - Simplify control unit to reduce power
  - Exploit ILP to improve performance

DSP: Power reduction

- Simple control
  - In-order
  - No HW interlocks (i.e. no HW hazard detection)
  - “Exposed pipeline”: compiler explicitly marks parallel instructions and inserts nops as needed
- Non-orthogonal ISA
  - Different opcodes support different register subsets
- Lack of caches:
  - Explicit tightly-coupled (AKA local, AKA scratchpad) memory instead of cache
  - Access to external memory stalls the core (organizing efficient data upload to local memory is a important part of DSP program)
  - Delay slots, branch hints, zero-overhead loops instead of BTB

DSP: ILP increase

- Explicitly parallel instructions
- SIMD (vector operations)
- Complex instructions
  - Heavy math: MACs, sqrt/isqrt, complex numbers, histograms, etc.
  - Post-shifter (important for fixed-point multiply)
  - Rich addressing modes (strided pre/post-modifications, cyclic/bit-reversed addressing)
  - Domain-specific instructions (e.g. Hexagon has IP checksum, H.264 decode)
  - Extensible ISAs (Tensilica, CEVA)
- Multi-banked memory (through multiple parallel loads/stores)
- Reduce branch overheads
  - Full predication
  - Delay slots
  - Zero-overhead loops
- Fixed-function units for computationally-intensive algorithms (Viterbi, CDNN, FFT, QR)
- Offload periodic data transfers from core via complex DMA blocks

DSP classification

- Ultra low-power
  - Example: TI C28x
  - uC with a single-cycle MAC
  - 50 MHz
  - Lots of peripherals
  - Standby mode(s)
- Low-end (“traditional”)
  - Example: TI C55x
  - No VLIW (or limited 2-way) or SIMD
  - 0.1 GHz, 0.1 W
  - Standby mode(s)
  - Less compiler friendly
- High-end (“modern”)
  - Example: TI C64x
  - Aggressive VLIW (5-8 way) and SIMD
  - 0.5-1 GHz, 0.5-5 W
  - Compiler-only

Low-end (“traditional”) DSPs

- Killer features:
  - Single-cycle MAC
  - HW loops (“zero-overhead loops”)
  - Address postmodification
  - Multiple loads per cycle (multiple memory banks)
- Execute filtering step in 1 cycle:

```
// This loop ...
for (i = 0; i < N; ++i)
  y += h[i] * x[i];
// ... compiles to a single complex instruction:
RPT AR5 || MAC R7, R3, *XAR6++, *XAR7++
```

High-end (“modern”) DSPs

- Revolutionary change in 5-th DSP generation (ca. 2000)
- Caused by:
  - Integration level advances (Moore’s law)
  - New markets
  - Increased complexity and size of DSP applications (over 10 KLOC assembly)
  - TTM constraints
- Transition to compiler-friendly VLIW and/or SIMD architectures
  - Also major compiler improvements in compiler theory (thanks to Itanium’s Open64 compiler)
- Loop from previous slide:

```
loop:
  BR delay1 .CU loop || LD .LS0 (A0)+, R0 || LD .LS1 (A1)+, R1
  MAC R0, R1, R2
```

VLIW in DSPs (1)

- Very Long Instruction Word architectures (parallel instructions are explicitly marked by compiler/developer in code)
- Traditional VLIW disadvantages which plagued Itanium/Transmeta are not (or less) critical for DSP
- Lack of static ILP in typical apps: unlimited ILP in typical DSP program
- Stalling on cache misses: regular data structures, scratchpad memory, users (or vendor) willing to rewrite/annotate apps to squeeze performance
- Do not need “heroic compilers” anymore
  - Compiler technology has advanced a lot
  - Several open-source solutions are available
- No need for binary compatibility and legacy ISA support
  - Apps are always rebuilt (and often rewritten) for new devices
- Imprecise static branch prediction: users are fine with providing profile data
- To a lesser extent same arguments apply to GPU accelerators
  - Nvidia GeForce FX and now, pre-2011 AMD, Mali use VLIW architectures

VLIW in DSPs (2)

- Statically scheduled AKA exposed pipeline: in-order, no register renaming, stall on cache miss
- Exposed uarch (instruction latencies, delay slots)
- Homogeneous registers
- Most ISA is RISC
  - Complex and CISC instructions to save space
- HW loops
- Multi-banked memory
- Combined with SIMD (and uSIMD)
- Very hard to debug
  - Single-stepping not available

VLIW vs SIMD (ILP vs DLP)

- Modern VLIWs are typically combined with SIMD
- Allow to express more parallelism than pure in-order SIMD:

```
for(i = 0; i < N; ++i) {
  z[i] = a*x[i] + b*y[i];
}

// VLIW saves 2 cycles due to parallelism
vload (x)+, vx || vload (y)+, vy
vmuls ra, vx, vx || vmuls ra, vy, vy
vadd vx, vy, vz
vstore vz, (z)+
```

DSP: ecosystem

- DSP integration:
  - Discrete (TI, Qualcomm)
  - IP (Cadence/Tensilica, CEVA)
- Important peripherals:
  - Often paired with uC e.g. ARM in TI OMAP, RISCV in CEVA Wifi SoCs (system split to control and data “planes”)
  - Include sophisticated DMA for efficient data supply
  - Often combined with domain-specific accelerators (e.g. NN, Viterbi, multimedia codecs)
- Algorithms originally developed and investigated in MATLAB/Simulink (using floating point)

DSP: benchmarks

- Operation-level (microbenchmarks)
  - GMACs (critical for marketing)
  - Sqrts/sec.
  - Various data sizes (32x32, 16x16, etc.)
- Kernel-level
  - Most typical DSP loops
  - FIR/IIR, FFT, DCT, Viterby, GEMM
  - BDTImark2000 family (also dedicated benchmarks for video and communications)
  - EEMBC (former EDN) Telebench
- Full applications
  - Standard embedded benchmarks (mainly EEMBC Coremark)

DSP: tools

- Tools are special as well
  - Standard components: assemblers, optimizing linkers, profilers, various types of simulators (cycle-accurate, functional), compilers
  - Special components: power profilers, linear/optimizing assemblers, boilerplate code generators (RPC calls, etc.), host intrinsic libraries, autotuners, etc.
  - This talk focuses on compilers
- (Relatively) simple programs
  - Small size
  - May reliably estimate performance at compile-time
- More (much more) resources allocated for optimization:
  - Spend much more time on tuning (mainly loops)
  - Allow more complex flows (feedback-driven compilation, analyse optimization remarks from compiler)
  - Kernels (re)written with compiler intrinsics (or even assembler), pragmas and restrict/noalias annotations
  - Allow longer compile times for LTO or parameter sweep
- Need more control over tools behaviour
  - Normally less attention to “fancy” technologies like autovec
- No binary compatibility expected
  - No ISA compatibility for higher-end DSPs (all code in high-level languages)

Retargetable Estimation Scheme for DSP Architecture Selection

DSP compilers: overview

- Extremely important for performance and overall product success
  - Also for experimenting with architecture decisions at design time
  - 10x performance difference of -O3/-O0 is not unusual (vs typical 2-3x on desktops)
  - Manual assembly programming not possible
- Mainly based on open-source offerings:
  - Open64 – open-sourced Itanium compiler (dead, used by Tensilica/Cadence and others)
  - GCC (used by TI and old Hexagons)
  - LLVM (new CEVA and Hexagon cores)
- Multilevel intermediate representations (IR): AST, SSA, RTL
  - Compared to GPPUs, much more work has to be done at RTL-level
- Need much more precise modelling of architecture (pipeline, resource conflicts, instruction sizes, etc.)
  - Flaw in model will cause invalid code generation
- Main goal is to extract maximum ILP from user's code
  - Hide data dependencies via SW register renaming, loop unrolling and SW pipelining
  - Hide control dependencies via delay slots, predication and speculation
  - Break memory dependencies through special aliasing rules
- Most companies have moved (or are moving) to LLVM
  - Corporate-friendly license
  - Reduce frontend development costs
  - OpenCL and Halide support

Compiler intrinsics

- Special functionality exposed to user as functions:

```
int32x4_t acc;
int *p, coeff;
for (i = 0; i < N; I += 4) {
  int32x4_t x = vload(&p[i]);
  acc = vmac(acc, coeff, x);
}
```

- Provide access to most of ISA (at least vector ISA)
- Easier to write than inline (or outline) assembly due to automatic register allocation, scheduling and packetizing
  - Assembler is still widely used, to the point that TI and QC ship optimizing assemblers in their toolchains
- Often also provides better performance (due to compiler optimizations like SWP or CSE over intrinsics)
- Most natural way for exploiting dedicated instructions
- Allows for (limited) source-level compatibility with old cores
- Disadvantage: vendor lock-in

Compiler pragmas

- TI compiler pragmas (same for other vendors)
  - #pragma MUST_ITERATE(min, max, multiple)
  - #pragma FUNC_NO_GLOBAL_ASG(func)
  - #pragma FUNC_NO_IND_ASG(func)
  - _nassert(assume_this_condition_is_true);
  - Etc.

Language extensions: strict*er* aliasing

- Relieve users from manually annotating pointers with “restrict”s
- Example: TI’s --no-bad-aliases
- Non-standard (stricter) aliasing rules
  - Pointers do not alias globals
  - Functions do not cache pointer arguments
  - Distinct source-level pointer variables can not alias

TI Compiler UG: Indicating Whether Certain Aliasing Techniques Are Used

Compiler optimizations: software pipelining

- A critical VLIW optimization
- Depends on other transformations to linearize loop bodies (unroll, inlining, data speculation, if-conversion, IV rename)
- Coupled with loop unrolling
- Compare two loops:

```
for (i = 0; i < N; ++i)
  a[i] = b[i] * 3;
```

```
tmp2 = b[0] * 3;
tmp1 = b[1];
for (i = 0; i < N - 2; ++i) {
  a[i] = tmp2
  tmp2 = tmp1 * 3;
  tmp1 = b[i + 2]
}
a[N - 2] = tmp2;
a[N - 1] = tmp1 * 3;
```

Compiler optimizations: speculation

- Boosts ILP at the cost of code size
- Depends on good frequency estimation

```
x = *p;
nop;
y = x + t;
if (cond)
  z = y * 3;
```

```
x = *p;
if (cond) {
  y = x + t;
  z = y * 3;
} else {
  // Compensation code
  y = x + t;
}
```

Compiler optimizations: if-conversion

- Removes complex control flow at the cost of higher power consumption
- Depends on good frequency estimation

```
for (i = ...) {
  if (p[i])
    x[i] = a * y;
  else
    x[i] = b * z;
}
```

```
for (i = ...) {
  tmp1 = a * y;
  tmp2 = b * z;
  x[i] = p[i] ? tmp1 : tmp2;
}
```

Compiler optimizations: IV renaming

- Removes false dependencies on address increments at the cost of higher register pressure

```
for (i = ...) {
  z[i] = a * x[i]
  ++i;
  z[i] = a * x[i];
  ++i;
}
```

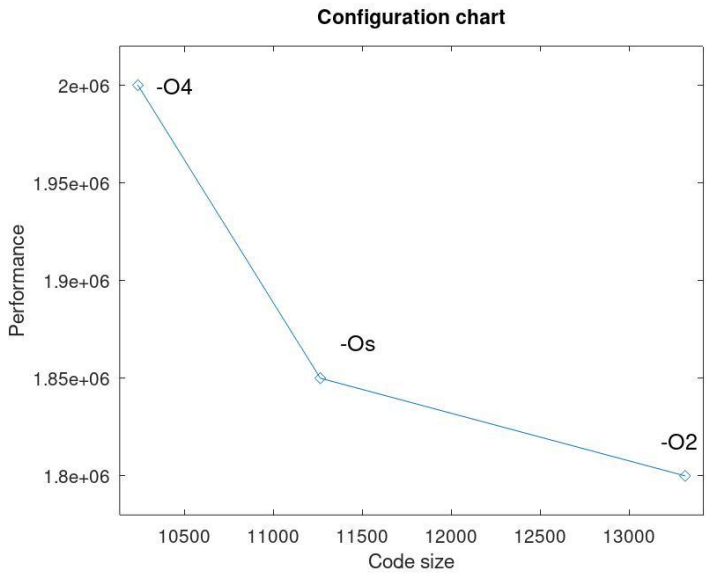
```
for (i1, i2 = ...) {
  z[i1] = a * x[i1]
  ++i1;
  z[i2] = a * x[i2];
  ++i2;
}
```

Compiler optimizations: optimization remarks

- All important optimization phases can produce compiler dumps with information on why optimization failed
- Information integrated to generated assembly code and IDE
- Optimization hints with suggestions for code optimization (adding pragmas, potentially redundant dependencies, etc.)
  - No autofixes in production yet

Compiler optimizations: parameter sweep (“autotuning”)

- DSP applications are typically smaller than general purpose ones
- Users are fine with trading compile time for better performance
- Break hard compilation problems with brute-force
  - State-space exploration of compiler parameters
  - Trying different unroll/SWP factors, tile/vector sizes, etc.
  - Static estimation of generated assembly
- Can be done in
  - Compiler
  - IDE
  - External tool (“auto-tuner”)



[Autotuning in HPC Applications]

Example DSP compiler: Open64

- Based on SGI MIPSpro compiler for MIPS, later ported to Itanium
- Later open-sourced and adopted by various VLIW shops (Nvidia, Tensilica, etc.)
- Used by Pathscale in their highly-optimizing HPC compiler
- Uses GCC C/C++ frontend (=> stuck in GCC 4.2 C++98 times due to GPLv3)
- Split to multiple phases (with dedicated executable per phase):
  - Frontend
  - Inliner
  - SSA optimizer
  - Backend
- Highly-influential at it's time (some decisions adopted by GCC/LLVM):
  - Dedicated IRs for front, middle and back-ends
  - Multi-level middle-end IR (from very high-level to machine-oriented)
  - Memory SSA
  - SSA-based partial redundancy elimination and induction variable optimization
  - Register allocation via prioritized coloring

Development of an Efficient DSP Compiler Based on Open64

Example DSP compiler: LLVM Hexagon

- One of the largest non-mainstream backends in LLVM
  - Largest intrinsics definition file
- Custom scheduler and VLIW packetizer
  - LLVM VLIW representation added specifically for Hexagon
  - Very strict and precise scheduling rules (due to exposed pipeline)
  - Non-data hazards represented via DAG mutations
- A lot of new generic data structures and optimization passes in MIR:
  - SSA bit tracker
  - Bit-level symbolic evaluator
  - Post-SSA reaching definitions
  - Generic MIR constant propagation/algebraic simplification
  - SSA if-converter (default converter is post-RA)
  - Predicator
  - Address mode optimizer
  - Mask operations generator
  - Load/store widener
- Most of above in lib/Target/Hexagon
  - Plans to merge to lib/Codegen for general use (maybe, some day...)
- Stock LLVM does not provide enough support for MIR optimization
  - Most work is done on LLVM IR

K. Parzyszek. Register Data Flow Framework

DSP: past trends

- Increase performance/power (GMACS/W)
  - Gene's law: GMACS/W doubles every 18 months
  - Vector length increase
  - Less arch innovation (main arch features available for decades)
- Improve productivity/usability
  - Higher-level languages (assembler to C and now C++)
  - Autovectorization
  - Optimization assistants
  - Floating-point support
  - Using IDE for automation of complex tasks (profile-guided optimization, etc.)
  - Simplify integration with customer's accelerators
- Specialization
  - Different chips (ISAs) for different markets (see CEVA product lines or Tensilica Customizable Processors)
  - Custom built-in accelerators

GMACS = MACs/second

DSP: current/future trends

- Increased arch and compiler innovation due to expiration of Moore/Gene's law
  - Arch
    - Multicore DSPs
    - Heterogeneous memories (with sophisticated DMAs to manage transfers)
    - Extensible ISAs, domain-specific accelerators, automatic ISA generation (Tensilica)
    - Register clustering
  - Compiler improvements
    - Optimization/analysis improvements (alias analysis, autovec, whole-program optimizations)
    - Generic backend optimizations (SWP, peepholes, if-conversion, etc.)
    - Support for DSP features (native complex types, HW loops, etc.)
    - Autotuning (Halide, TVM, etc.)
  - Improvements in other tools
    - Support for standard libraries (OpenCV, Eigen, etc.), frameworks (Tensorflow, Caffe, TVM, etc.), languages (OpenCL, Halide) and development platforms (PX4, Dronecode, GNURadio, etc.)
    - Autofix support in assistants
    - Improved debugging experience
    - Code generation to simplify redundant work (Hexagon FastRPC for offloading, TI C2x peripheral customizations, etc.)
- New markets
  - Neural networks
  - IoT, EDGE, etc.
  - Control MCUs (CEVA BX for Edge computing)
  - HPC (Matrix-2000 GPDSP in China's Tianhe-2)
  - Bioinformatics (?)
  - Etc.

Further reading

- General VLIW:
  - **J.A. Fisher et al. “Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools”**
  - “Mill computing” series of lectures on Youtube (and their site https://millcomputing.com)
- VLIW compilers:
  - P.G. Lowney The Multiflow Trace Scheduling Compiler
  - The Making of a Compiler for the Intel Itanium Processor
  - DSP-C Specification and Embedded-C extensions
  - TI compiler manuals (User and Programmer guides)
- CEVA arch:
  - CEVA Launches Machine Learning DSP Solution: CEVA-XM6 (anandtech)
  - CEVA-XC12 The world's most advanced communications DSP
- CEVA compilers:
  - E. Belaish Combining C code with assembly code in DSP applications
  - E. Belaish Architecture Oriented C Optimizations
  - E. Belaish Compiler optimization for DSP applications

The End

DSP vs general-purpose processors (GPPU)

- Intel Skylake Xeon Platinum 8180M
  - Frequency: 2.5 GHz
  - Cores: 28
  - Peak issue width: 8
  - Peak performance: 560 GIPS
  - Power: 205 W
  - Price: $13K (plus coolant!)
  - Perf/W/$/core: **7.5 KIPS/W/$/core**
- TI TMS320C6713BZDP300
  - Frequency: 300 MHz
  - Cores: 1
  - Peak issue width: 6
  - Peak performance: 1.8 GIPS
  - Power: around 1 W
  - Price: $35 (for 1KU)
  - Perf/W/$/core: **51 MIPS/W/$/core (7000x over Xeon!)**
- Caveat:
  - DSPs only achieve peak performance on highly-regular loops and data structures.

Drinking from the Firehose: How the Mill CPU Decodes 30+ Instructions per Cycle

Changelog

- 0.3
  - First public version
- 0.4
  - Multi-banked memory in “Modern VLIW architectures”
  - Host emulators in “DSP: tools”
- 0.5
  - Final version for Samsung
- 0.6
  - Update content