Rechnerarchitektur (RA)

Sommersemester 2020

Architecture-Aware Optimizations
- Hardware-Software co-Optimizations-

Jian-Jia Chen
Informatik 12
Jian-jia.chen@tu-..
Tel.: 0231 755 6078
Outline

High-Level Optimization
- Loop transformation
- Loop tiling/blocking
- Loop (nest) splitting

Heterogeneous System Architecture (HSA)
- Integrated CPU/GPU platforms
- Recent movement in chip designs

Architecture-aware software designs
- Energy-efficiency issues
- Darkroom
- Halide

Multicore revolutions
- Impact on the “safety-critical” industry sector
Impact of memory allocation on efficiency

Array $p[j][k]$

Row major order (C)

$$
\begin{align*}
  & j=0 \\
  & \quad k=0 \\
  & \quad k=1 \\
  & \quad \ldots \\
  & j=1 \\
  & \quad k=0 \\
  & \quad k=1 \\
  & \quad \ldots \\
  & j=2 \\
  & \quad k=0 \\
  & \quad k=1 \\
  & \quad \ldots \\
  & \vdots
\end{align*}
$$

Column major order (FORTRAN)

$$
\begin{align*}
  & k=0 \\
  & \quad j=0 \\
  & \quad j=1 \\
  & \quad \ldots \\
  & k=1 \\
  & \quad j=0 \\
  & \quad j=1 \\
  & \quad \ldots \\
  & k=2 \\
  & \quad j=0 \\
  & \quad j=1 \\
  & \quad \ldots \\
  & \ldots
\end{align*}
$$
Best performance of innermost loop corresponds to rightmost array index

Two loops, assuming row major order (C):

\[
\begin{align*}
\text{for } & (k=0; k<=m; k++) & \text{for } & (j=0; j<=n; j++) \\
\text{for } & (j=0; j<=n; j++) & \text{for } & (k=0; k<=m; k++) \\
p[j][k] & = ... & p[j][k] & = ...
\end{align*}
\]

Same behavior for homogeneous memory access, but:

For row major order

\[\uparrow \text{Poor cache behavior} \quad \text{Good cache behavior} \uparrow\]

\[\text{memory architecture dependent optimization}\]
Program transformation “Loop interchange”

Example:
...
#define iter 400000
int a[20][20][20];
void computeijk() {int i,j,k;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            for (k = 0; k < 20; k++) {
                a[i][j][k] += a[i][j][k];}}}}
void computeikj() {int i,j,k;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            for (k = 0; k < 20; k++) {
                a[i][k][j] += a[i][k][j];}}}}

 Improved locality

...start= time(&start);for(z =0;z< iter;z++)computeijk();
end= time(&end);
printf("ijk =\%16.9f \n",1.0* difftime(end,start));
Results:
strong influence of the memory architecture

Loop structure: i j k

Dramatic impact of locality

<table>
<thead>
<tr>
<th>Processor</th>
<th>Ti C6xx</th>
<th>Sun SPARC</th>
<th>Intel Pentium</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduction to [%]</td>
<td>~ 57%</td>
<td>35%</td>
<td>3.2 %</td>
</tr>
</tbody>
</table>

Not always the same impact..

Transformations
“Loop fusion” (merging), “loop fission”

\[
\text{for}\ (j=0; \ j<=n; \ j++) \\
p[j]= ... ; \\
\text{for}\ (j=0; \ j<=n; \ j++) , \\
p[j]= p[j] + ... \\
\]

Loops small enough to allow zero overhead

Better locality for access to p.
Better chances for parallel execution.

Which of the two versions is best?
Architecture-aware compiler should select best version.
Example: simple loops

```c
#define size 30
#define iter 40000
int a[size][size];
float b[size][size];

void ss1() {int i,j;
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            a[i][j]+= 17;
        }
    }
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            b[i][j] -= 13;
        }
    }
}

void ms1() {int i,j;
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            a[i][j] += 17;
        }
    }
    for (j=0;j<size;j++) {
        b[i][j] -= 13;
    }
}

void mm1() {int i,j;
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            a[i][j] += 17;
            b[i][j] -= 13;
        }
    }
}

#define size 30
#define iter 40000
int a[size][size];
float b[size][size];

void ss1() {int i,j;
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            a[i][j]+= 17;
        }
    }
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            b[i][j] -= 13;
        }
    }
}

void ms1() {int i,j;
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            a[i][j] += 17;
        }
    }
    for (j=0;j<size;j++) {
        b[i][j] -= 13;
    }
}

void mm1() {int i,j;
    for (i=0;i<size;i++) {
        for (j=0;j<size;j++) {
            a[i][j] += 17;
            b[i][j] -= 13;
        }
    }
}
```

```c
```
Results: simple loops

Merged loops superior; except Sparc with –o3
Loop unrolling

\begin{align*}
\textbf{for} \ (j=0; \ j<=n; \ j++) \ &
\begin{array}{ll}
p[j] = \ldots \ ;
\end{array}
\end{align*}

\rightarrow

\begin{align*}
\textbf{for} \ (j=0; \ j<=n; \ j+=2) \ &
\begin{array}{ll}
\{p[j] = \ldots \ ; \ p[j+1] = \ldots\}
\end{array}
\end{align*}

factor = 2

\textit{Better locality} for access to \(p\).
Less branches per execution of the loop. More opportunities for optimizations.

\textit{Tradeoff between code size and improvement.}

\textit{Extreme case: completely unrolled loop (no branch).}
Program transformation
Loop tiling/loop blocking: - Original version -

```plaintext
for (i=1; i<=N; i++)
  for (k=1; k<=N; k++){
    r=X[i,k]; /* to be allocated to a register*/
    for (j=1; j<=N; j++)
      Z[i,j] += r* Y[k,j]
  }
% Never reusing information in the cache for Y and Z if N is large or cache is small (O(N^3) references for Z).
```

![Diagram](image)
Loop tiling/loop blocking
- tiled version -

```
for (kk=1; kk< N; kk+=B)
  for (jj=1; jj< N; jj+=B)
    for (i=1; i< N; i++)
      for (k=kk; k< min(kk+B-1,N); k++)
        r=X[i][k]; /* to be allocated to a register*/
        for (j=jj; j< min(jj+B-1, N); j++)
          Z[i][j] += r* Y[k][j]
```

Reuse factor of
B for Z, N for Y

$O(N^3/B)$
accesses to
main memory

Compiler should select
best option

Transformation “Loop nest splitting”

Example: Separation of margin handling

many if-statements for margin-checking

no checking, efficient

only few margin elements to be processed
Outline

High-Level Optimization
- Loop transformation
- Loop tiling/blocking
- Loop (nest) splitting

Heterogeneous System Architecture (HSA)
- Integrated CPU/GPU platforms
- Recent movement in chip designs

Architecture-aware software designs
- Energy-efficiency issues
- Darkroom
- Halide

Multicore revolutions
- Impact on the “safety-critical” industry sector
What is Heterogeneous Computing?

Use processor cores with various type/computing power to achieve better performance/power efficiency.

Advantage of Heterogeneous Computing

CPU is ideal for scalar processing
- Out of order x86 cores with low latency memory access
- Optimized for sequential and branching algorithms
- Runs existing applications very well

Serial/Task-parallel workloads $\rightarrow$ CPU

GPU is ideal for parallel processing
- GPU shaders optimized for throughput computing
- Ready for emerging workloads
- Media processing, simulation, natural UI, etc.

Graphics/Data-parallel workloads $\rightarrow$ GPU

Heterogeneous Computing $\rightarrow$ Fusion, Norm Rubin, SAAHPC 2010
CPU/GPU Integration: CPU’s Advancement Meets GPU’s

Microprocessor Advancement
- Single-Thread Era
- Multi-Core Era
- Heterogeneous Systems Era

High Performance Task Parallel Execution

Heterogeneous Computing
- System-Level Programmable
- Vertex/Pixel Shader Graphics Driver-based programs
- Power-efficient Data Parallel Execution

Throughput Performance

Programmability
- Unacceptable
- Experts Only
- Mainstream
Evolution of Heterogeneous Computing

Dedicated GPU
- GPU kernel is launched through the device driver
- Separate CPU/GPU address space
- Separate system/GPU memory
- Data copy between CPU/GPU via PCIe

![Diagram of GPU architecture]

- OpenCL Application
- OpenCL Runtime Library
- User Space
- Kernel Space
- GPU Device Driver
- Address space managed by OS
- Address space managed by driver
- CPU Core1, Core2, CoreN
- Core1 L1/L2, Core2 L1/L2, CoreN L1/L2
- LLC
- System memory (coherent)

- GPU CU1, CU2, CUN
- CU1 L1, CU2 L1, CUN L1
- GPU memory (Non-coherent)

= kernel launch process
**Evolution of Heterogeneous Computing**

**Integrated GPU architecture**
- GPU kernel is launched through the device driver
- Separate CPU/GPU address space
- Separate system/GPU memory
- Data copy between CPU/GPU *via memory bus*

---

**Diagram**
- GPU kernel launch process
- Data preparation
- Computation

**Key Points**
- GPU kernel launch
- Separate address space
- Separate memory spaces
- Data copy via memory bus
Evolution of Heterogeneous Computing

Integrated CPU/GPU architecture

- GPU kernel is launched through the device driver
- Unified CPU/GPU address space (managed by OS)
- Unified system/GPU memory
- No data copy - data can be retrieved by pointer passing

```
GPU kernel is launched through the device driver
Unified CPU/GPU address space (managed by OS)
Unified system/GPU memory
No data copy - data can be retrieved by pointer passing
```
Utopia World of Heterogeneous Computing

Processors are architected to operate cooperatively
- Tasks in an application are executed on different types of core
- Unified coherent memory enables data sharing across all processors

Designed to enable the applications to run on different processors at different time
- Capability to translate from high-level language to target binary at run-time
- User-level task dispatch
- Decision making module

Coherent system memory
HSA Foundation

Founded in June 2012
Developing a new platform for heterogeneous systems
www.hsafoundation.com
Specifications under development in working groups to define the platform
Diverse Partners Driving Future of Heterogeneous Computing

Founders
- AMD
- ARM
- Imagination
- MediaTek
- Qualcomm
- Samsung
- Texas Instruments

Promoters
- LG Electronics
- Arteris
- FABRICEngine
- Lawrence Livermore National Laboratory
- Allinea
- Argonne

Supporters
- codeplay
- MulticoreWare
- Linaro
- Sandia National Laboratories
- SUSE
- Oak Ridge National Laboratory
- Stream Computing Performance Engineers

Contributors
- Analog Devices
- Apical
- Broadcom
- Canonical
- CEVA
- ETRI
- Industrial Technology Research Institute
- Marvell
- Mobica
- OpenMP
- OpenCL
- Sonics
- Sony
- Synopsys
- tensilica
- Toshiba
- Tux
- Ubuntu
- Vivante
- SB Graphics
- Oracle
HSA (Heterogeneous System Architecture) Hardware-Software Stack

Application

OpenCL

Task 1
Task 2
Task 3

OpenCL Compiler

HSA Intermediate Language (HSAIL)

Task 1
Task 2
Task 3

HSA Run-time

Agent Scheduler

CPU Finalizer

GPU Finalizer

Native machine codes

Core 1
Core 2

CU 1
CU 2

AC 1
AC 2

Shared virtual address space

User-level SW

→

HW

→
Intel Haswell

Haswell Processor Family Overview (Traditional)

- 22nm Process
- Socket: G3 (947 pin) MB, H3 (1150 pin) DT
- Fully Integrated VR
- Power Aware Interrupt Routing for power/ performance
- Intel® Hyper-Threading Technology
- Intel® AVX 2.0 extensions and AES-NI Instructions Improvements
- Intel® Turbo Boost Technology

- Processor Graphics
  - 3D GFX/Media Arch/Perf.
  - Support for new APIs
  - 5 Digital Displays
  - DP 1.2/HDMI 1.4/eDP

- Desktop/All-In-One
  - DDR3/DDR3L-1600 2D/Ch non-ECC UDIMM/SODIMM

- Mobile
  - DDR3L (only)
  - POR for 1600 2D/Ch 6L rPGA non-ECC SODIMM
  - DDR Power Gating for lower Idle power (MB)
  - PCIe 3.0 x16/2x8 3 controllers
  - Power Optimizer (CPPM) Support (MB)

Haswell Offers Better Power & Performance with Improved I/O
AMD Kaveri APU

“KAVERI” FEATURING UP TO 12 COMPUTE CORES (4 CPU + 8 GPU)

CPU COMPUTE CORES

Up to four new multi-threaded AMD “Steamroller” CPU CORES

GPU COMPUTE CORES

Up to eight GCN GPU CORES® powering parallel compute and next-gen gaming

Visit amd.com/computecores for more detail
NVIDIA Tegra K1

Kepler GPU (192 CUDA Cores)
Open GL 4.4, OpenGL ES3.0, DX11, CUDA 6

Quad Core Cortex A15 “r3”
With 5th Battery-Saver Core; 2MB L2 cache

Dual High Performance ISP
1.2 Gigapixel throughput, 100MP sensor

Lower Power
28HPM, Battery Saver Core

4K panel, 4K HDMI
DSI, eDP, LVDS, High Speed HDMI 1.4a
Qualcomm Snapdragon
Outline

High-Level Optimization
- Loop transformation
- Loop tiling/blocking
- Loop (nest) splitting

Heterogeneous System Architecture (HSA)
- Integrated CPU/GPU platforms
- Recent movement in chip designs

Architecture-aware software designs
- Energy-efficiency issues
- Darkroom
- Halide

Multicore revolutions
- Impact on the “safety-critical” industry sector
Energy Efficiency of different target platforms

© Hugo De Man, IMEC, Philips, 2007
Signal Processing ASICs

Markovic, EE292 Class, Stanford, 2013
How about Memory?
Processor Energy with Corrected Cache Sizes

© Horowitz, DAC 2016
Processor Energy Breakdown

© Horowitz, DAC 2016
Data Center Energy Specs

© Malladi, ISCA 2012
What Is Going On Here?

Markovic, EE292 Class, Stanford, 2013
Energy Consumption (Approximate, 45nm)

<table>
<thead>
<tr>
<th>Integer</th>
<th></th>
<th>FP</th>
<th></th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td></td>
<td>FAdd</td>
<td>16 bit</td>
<td>Cache (64bit)</td>
</tr>
<tr>
<td>8 bit</td>
<td>0.03pJ</td>
<td>0.4pJ</td>
<td>8KB</td>
<td>10pJ</td>
</tr>
<tr>
<td>32 bit</td>
<td>0.1pJ</td>
<td>0.9pJ</td>
<td>32KB</td>
<td>20pJ</td>
</tr>
<tr>
<td>Mult</td>
<td></td>
<td>FMult</td>
<td>16 bit</td>
<td>1MB</td>
</tr>
<tr>
<td>8 bit</td>
<td>0.2pJ</td>
<td>1pJ</td>
<td>DRAM</td>
<td>1.3-2.6nJ</td>
</tr>
<tr>
<td>32 bit</td>
<td>3 pJ</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Energy Breakdown

© Horowitz, DAC 2016
True Story

It’s more about the algorithm than the hardware.
The efficiency cannot be achieved unless the algorithm is right!!

(all) Algorithms

GPU Alg.
Locality, Locality, and Locality!!!
Darkroom (Stanford/MIT)

© Hegarty et al., SIGGraph 2014
Locality versus Parallelism

Halide Programming Language:

- http://halide-lang.org/

Performance needs a lot of tradeoffs

- Locality
- Parallelism
- Redundant recomputation
Outline

High-Level Optimization
- Loop transformation
- Loop tiling/blocking
- Loop (nest) splitting

Heterogeneous System Architecture (HSA)
- Integrated CPU/GPU platforms
- Recent movement in chip designs

Architecture-aware software designs
- Energy-efficiency issues
- Darkroom
- Halide

Multicore revolutions
- Impact on the “safety-critical” industry sector
Automotive Software

Task

- Activation Pattern:
  - Periodic: 1 to 1000 ms
  - Angle synchronous
  - Sporadic

- Scheduled by the OS
  - Fixed Priorities
  - Preemptively or cooperatively

© Hamann, Kramer, Ziegenbein, Lukasiewycz (Bosch), 2016
Assessment of Multi-Core Worst-Case Execution Behavior

Communication between runnables is realized with reading and writing of labels

<table>
<thead>
<tr>
<th>Communication</th>
<th>Share</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>25 %</td>
</tr>
<tr>
<td>Backward</td>
<td>35 %</td>
</tr>
<tr>
<td>InterTask</td>
<td>40 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</th>
<th>Share</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic (1-4 bytes)</td>
<td>97 %</td>
</tr>
<tr>
<td>Structs / Arrays</td>
<td>3 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Access type</th>
<th>Share</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-only</td>
<td>40 %</td>
</tr>
<tr>
<td>Write-only</td>
<td>10 %</td>
</tr>
<tr>
<td>Read-Write</td>
<td>50 %</td>
</tr>
</tbody>
</table>

© Hamann, Kramer, Ziegenbein, Lukasiewycz (Bosch), 2016
Multi-Core Memory Access Models

Access time to data in different memories (local & global)

© Hamann, Kramer, Ziegenbein, Lukasiewycz (Bosch), 2016
An Industrial Challenge (FMTV 2017)

Precise analysis of worst-case end-to-end latencies

- mainly due to different involved periods and time domains
- What is the effect on memory layout and interconnect on the execution times?
- Automatic optimized application and data mapping
- Evaluation of digital (multi-core) execution platforms
- Evaluation of software growth scenarios

© Hamann, Kramer, Ziegenbein, Lukasiewycz (Bosch), 2016
MPPA-256 Processor Architecture (Kalray)

- 16 compute clusters
- 2 I/O clusters each with quad-core CPUs, DDR3, 4 Ethernet 10G and 8 PCIe Gen3
- Data and control networks-on-chip
- Distributed memory architecture
- 634/317 GFLOPS SP/DP @ 600Mhz

- 16 user cores + 1 system core
- NoC Tx and Rx interfaces
- Debug & Support Unit (DSU)
- 2 MB multi-banked shared memory
- 77GB/s Shared Memory BW
- 16 cores SMP System

- 32-bit or 64-bit addresses
- 5-issue VLIW architecture
- MMU + I&D cache (8KB+8KB)
- 32-bit/64-bit IEEE 754-2008 FMA FPU
- Tightly coupled crypto co-processor
- 2.4 GFLOPS SP per core @600Mhz

Kalray, 2016
MPPA-256 NoC

- Dual 2D-torus NoC
  - D-NoC: high-bandwidth RDMA
  - C-NoC: low-latency mailboxes
  - 4B/cycle per link direction per NoC
  - Nx10Gb/s NoC extensions for connection to FPGA or other MPPA®

- Predictability
  - Data NoC is configured by selecting routes and injection parameters
  - Injection parameters are the \((\sigma,\rho)\) or (burst, rate) of Cruz network calculus
  - Guaranteed services rely on same methods as in AFDX Ethernet
Safety-Critical Systems with Multicore Platforms

**Goal:** deploy multi-core processors for safety-critical real-time applications (avionics, automotive,…)

**Problem:** concurrent use of shared resources (e.g. interconnect, main memory)
- unknown access latency for a concrete resource access
- complicated timing analysis
- hardware platforms may not be predictable
- Many features are designed by computer architects for *average cases* only

**Solution?**
- Maybe it is up to you.
- Did you see the above challenges?