Lecture 1

Introduction

I. Why Study Compilers?
II. Example: Machine Learning
III. Course Syllabus

Chapters 1.1-1.5, 8.4, 8.5, 9.1
Reasons for Studying Program Analysis & Optimizations

- Implementation of a programming abstraction
  - Improves software productivity by hiding low-level details while delivering performance
  - Domain-specific languages

- A tool for designing and evaluating computer architectures
  - Inspired RISC, VLIW machines
  - Machines' performance measured on compiled code

- Techniques for developing other programming tools
  - Example: error detection tools

- Program translation can be used to solve other problems
  - Example: Binary translation (processor change, adding virtualization support)

  • Compilers have impact: affect all programs

Compiler Study Trains Good Developers

Excellent software engineering case study

• Optimizing compilers are hard to build
  - Input: all programs
  - Objectives:

• Methodology for solving complex real-life problems
  - Key to success: Formulate the right approximation!
    - Desired solutions are often NP-complete / undecidable
  - Where theory meets practice
    - Can't be solved by just pure hacking
      - theory aids generality and correctness
    - Can't be solved by just theory
      - experimentation validates and provides feedback to problem formulation

• Reasoning about programs, reliability & security makes you a better programmer
  
There are programmers, and there are tool builders ...
Example: Machine Learning

- **Machine learning**
  - Important to improve programmers’ productivity
    - Powerful, widely applicable
  - Many variants of common ideas
    - Need programmability
    - Amenable to languages + compiler techniques
  - GPUs are difficult to optimize

- **Demonstrate the importance of**
  - Programming languages: Programming abstraction
    - A beautiful hierarchy of abstractions
    - Bridge the gap from neural nets to GPUs
  - Compiler optimizations: Performance for high-level programming
    - Automatic analysis and transformation
    - Hide complex machine details
    - Provide machine independence

Outline

1. Why is optimizing machine-learning important and hard?
2. Today’s approach:
   - Programming abstractions + handcoded libraries
3. Need and opportunities for compiler optimizations
1. Machine Learning is Expensive

Microsoft ResNet Superhuman Image Recognition

Baidu Deep Speech 2 Superhuman Voice Recognition

Google Neural Machine Translation Near Human Language Translation

1 ExaFLOPS = 10^{18} FLOPS

Nvidia Volta GV100 GPU

21B transistors
815 mm²
1455 Mhz
80 Stream Multiprocessors (SM)

In Each SM

- 64 FP32 cores
- 64 int cores
- 32 FP64 cores
- 8 Tensor cores

Tensor Cores

\[ D = A \times B + C; \quad A, B, C, D \text{ are 4x4 matrices} \]

4 x 4 x 4 matrix processing array

1024 floating point ops / clock

FP32: 15 TFLOPS
FP64: 7.5 TFLOPS
Tensor: 120 TFLOPS

General algorithm

Training:
- Generate parameters with a large data set
- Start with initial values
- Iterate
  - Forward pass on labeled data
  - Back propagate

Inference:
- Forward pass with trained parameters

Note:
- Many popular architectures
- Common building blocks
Example: Inception – a Convolutional Neural Network for Object Recognition

https://towardsdatascience.com/transfer-learning-using-keras-d804b2e04ef8

Neural Networks to ExaFLOPS

2 Intel Xeon E5-2698 V4 processors
Each: 20 cores, 40 threads at 2.2 GHz

8 Nvidia Volta (960 TFLOPS)
Challenges

• Many applications of deep neural nets
  – Want to enable many developers to write efficient neural net code

• Can we transform a high-level neural net algorithm to
  – Efficient code for a complex target?
    Heterogeneous and distributed/parallel architectures:
    • x86 (multi-cores, SIMD instructions)
    • GPU (blocks of streams, Tensor cores)
  – For different architectures?
    • Architecture upgrades: e.g. Pascal to Volta (Nvidia)
    • Alternative architectures:
      Intel, AMD, TPU (tensor processing unit), FPGA
    • Training vs inference: servers to mobile devices

2. Today’s Solution

• Bridge the semantic gap from Neural Networks to GPUs
  via domain-specific abstractions
  – Computation model abstractions
  – Hand-optimized primitives
## Levels of Abstraction

<table>
<thead>
<tr>
<th>Neural Net Model</th>
<th>Inception</th>
<th>AlexNet</th>
<th>Seq2Seq</th>
<th>YOLO</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Level Framework</td>
<td>Keras</td>
<td>TFslim</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low Level Framework</td>
<td>Tensorflow</td>
<td>Caffe</td>
<td>MxNet</td>
<td>PyTorch</td>
</tr>
<tr>
<td>High Level Library</td>
<td>cuDNN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low Level Library</td>
<td>cuBLAS, cuFFT, cuSPARSE</td>
<td>Eigen</td>
<td>Open BLAS</td>
<td></td>
</tr>
<tr>
<td>GPU Language</td>
<td>CUDA</td>
<td>Open CL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td>Nvidia GPU</td>
<td>x86</td>
<td>AMD</td>
<td>Google TPU</td>
</tr>
</tbody>
</table>

## One Popular Stack

<table>
<thead>
<tr>
<th>Neural Net Model</th>
<th>Network of neural network modules (conv2d, VGG: convolutional NN for image recognition)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Level Framework</td>
<td>Keras</td>
</tr>
<tr>
<td>Low Level Framework</td>
<td>Tensorflow</td>
</tr>
<tr>
<td>High Level Library</td>
<td>cuDNN</td>
</tr>
<tr>
<td>Low Level Library</td>
<td>cuBLAS, cuFFT, cuSPARSE</td>
</tr>
<tr>
<td>GPU Language</td>
<td>CUDA</td>
</tr>
<tr>
<td>Hardware</td>
<td>GPU</td>
</tr>
</tbody>
</table>

Explicit CPU/GPU code + memory management
**Keras: Python framework to call a library of neural network modules**

```
x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv1')(img_input)
model = applications.VGG19(weights = "imagenet",
                         include_top=False,
                         input_shape = (img_width, img_height, 3))
```

**Tensorflow: Graph of Operations in a Neural Network Module**

```
W = tf.get_variable(name='W', shape=(K_in, K_out), dtype=tf.float32, trainable=True)
x = tf.placeholder(shape=(None, K_in), dtype=tf.float32)
y = tf.placeholder(shape=(None, K_out), dtype=tf.float32)
y_hat = tf.matmul(x, W)
y_predict = tf.sign(y_hat) > 0
j = tf.nn.sigmoid_cross_entropy_with_logits(logits=y_hat, labels=y)
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
optimization = optimizer.minimize(j)
```
Tensorflow: Execute the Flow Graph

Dynamically interpret the flow graph
- Processors have state: data and cached
- Assign flow graph nodes to processors
- Time: satisfying dependences
- Space: based on state or user preference
- Distributed machines: Generate receive & send codes

A user-level library that differentiates expressions to implement backpropagation automatically.

Management
- Fault tolerance
- Coordination and backup workers

Eigen: Open C++ Math Libraries

Open-source maths library
- Dense & sparse matrices
- Different data types
- Matrix operations, decompositions, …

Generates code for Nvidia (CUDA), x86, TPU, ...
- Explicit vectorization
- Optimizations for fixed-size matrices: e.g., loop unrolling

Implemented using expression templates
CuX: Proprietary Libraries in Cuda for Nvidia Processors

Carefully hand-tuned libraries for Nvidia architectures:
- cuBLAS: basic linear algebra subroutines
- cuFFT: fast Fourier transform
- cuSPARSE: BLAS for sparse matrices
- cuDNN: deep neural network
  - convolution, pooling, normalization, activation
  - enables optimizations across low-level libraries
Tuned for different matrix sizes, aspect ratios, architecture

CUDA: Programming Language from Nvidia

Programming model
- User controls the grids of threads in the GPU
- Manage memory

CUDA is a run-time system to control:
- Kernel dispatch to GPUs
- Synchronization
- Memory management
CUDA

```c
__global__ void add(int n, float *x, float *y)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}
```

```c
add<<<numBlocks, blockSize>>>(N, x, y);
```

```
ingridDim.x = 4096
```

```
threadIdx.x
0 1 2 3 255 0 1 2 3 255 0 1 2 3 255 ... 0 1 2 3 255

blockIdx.x = 0 1 2 3 255 0 1 2 3 255 0 1 2 3 255 ... 0 1 2 3 255

index = blockIdx.x * blockDim.x + threadIdx.x
index = (2) * (256) + (3) = 515
```

Levels of Abstraction

<table>
<thead>
<tr>
<th>Neural Net Model</th>
<th>Inception</th>
<th>AlexNet</th>
<th>Seq2Seq</th>
<th>YOLO</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Level Framework</td>
<td>Keras</td>
<td>TFslim</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low Level Framework</td>
<td>Tensorflow</td>
<td>Caffe</td>
<td>MxNet</td>
<td>PyTorch</td>
</tr>
<tr>
<td>High Level Library</td>
<td>cuDNN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low Level Library</td>
<td>cuBLAS</td>
<td>cuFFT</td>
<td>cuPARSE</td>
<td></td>
</tr>
<tr>
<td>GPU Language</td>
<td>CUDA</td>
<td>Open CL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td>Nvidia GPU</td>
<td>x86</td>
<td>AMD</td>
<td>Google TPU</td>
</tr>
</tbody>
</table>
3. Problems with Handcoded Libraries

- **Cost:** Expensive to write the tuned routines
- **Proprietary**
  - Nvidia proprietary libraries NOT available for other architectures
  - Competitive advantage that reduces innovation
- **Machine dependence**
  - Cannot generate code for different targets automatically
- **Performance:** Cannot optimize across library calls

- **Compilers**
  - XLA: Tensorflow $\rightarrow$ optimized code for different targets
  - TensorRT: trained model $\rightarrow$ inference engine (e.g. on CPU)

- **Optimization opportunities**
  - Cross-kernel optimization
  - Optimization of each kernel

---

Cross-Kernel Optimization: Fusion

$$s[j] = \text{softmax}[j] \ (\text{ReLU} \ (\text{bias}[j] + \text{matmul\_result}[j]))$$

https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html
**Optimizing Kernels: Matrix Multiply**

```c
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
            m3(i, j) += m1(i, k) * m2(k, j);
        }
    }
}
```

Permute loop to make data access contiguous for vectorization:

```c
for (k = 0; k < N; k++) {
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            m3(i, j) += m1(i, k) * m2(k, j);
        }
    }
}
```

**Tiling: to Increase Reuse**

```c
for (k = 0; k < N; k++) {
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            m3(i, j) += m1(i, k) * m2(k, j);
        }
    }
}
```

Tile the outermost loop

```c
for (k1 = 0; k1 < N; k1 += B) {
    for (i = 0; i < N; i++) {
        for (k2 = k1; k2 < k1 + B; k2++) {
            for (j = 0; j < N; j++) {
                m3(i, j) += m1(i, k2) * m2(k2, j);
            }
        }
    }
}
```

Assume N is divisible by B;
Optimal B (empirical): 128 bytes
Experiment

- Square float32 matrix of various sizes
- Initialized with random \((0, 1)\) normal
- Average of 10 iterations
- Intel i7-4770HQ CPU @ 2.20GHz (Haswell), with SSE4.2 and AVX2, no turbo
- 32k L1 cache, 256k L2, 6M L3, 132M L4 cache (LLC, GPU shared)
- Compiled with g++ 7.2.1 20170915, as provided in Fedora 27
- Common options: --std=c++14 -Wall -g
- (The production version of clang does not support loop optimizations)

Experiment variants

- Naive with no optimizations (-O0 -march=native)
- Naive with std optimizations (-O3 -march=native)
- Naive with full optimizations (-O3 -march=native -ftree-vectorize -floop-interchange -floop-nest-optimize -floop-block -funroll-loops)
- Permuted
- 1-d tiled with std optimizations
- 1-d tiled with full optimizations
### Performance

![Performance Graph](image)

### Optimizing the Matrix Multiply Kernel

```c
#pragma omp parallel for
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
            m3(i, j) += m1(i, k) * m2(k, j);
        }
    }
}
```

Permute loop to make data access contiguous:
```c
#pragma omp parallel
for (k = 0; k < N; k++) {
    #pragma omp for
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            m3(i, j) += m1(i, k) * m2(k, j);
        }
    }
}
```
Parallel 1-d tiled algorithm

#pragma omp parallel
for (k1 = 0; k1 < N; k1 += B) {
#pragma omp for
for (i = 0; i < N; i++) {
    for (k2 = k1; k2 < k1 + B; k2++) {
        for (j = 0; j < N; j++) {
            m3(i, j) += m1(i, k2) * m2(k2, j);
        }
    }
}}

m3 fetched N/B times

Parallel scaling (matrix size 1500)
Data Locality Optimization

- Data locality is important for sequential and parallel machines
  - Matrix multiplication has $N^2$ parallelism
  - Parallelization does not necessarily mean performance

- 2 important classes of loop transformations
  - Affine: Permutation, skewing, reversal, fusion, fission
  - Tiling

- Compilation
  - Machine-independence analysis
    - Is it legal to parallelize a loop? to transform a loop?
  - Machine-dependent optimization
    - which transformation to apply? for a certain machine

Loop Transformations

- Programs
  - static statements
  - dynamic execution
  - generated code

- Mathematical Model
  - matrices
  - integer programs
  - solutions

abstraction
Course Syllabus

1. Basic compiler optimizations

<table>
<thead>
<tr>
<th>Goal</th>
<th>Eliminates redundancy in high-level language programs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Allocates registers</td>
</tr>
<tr>
<td></td>
<td>Schedules instructions (for instruction-level parallelism)</td>
</tr>
<tr>
<td>Scope</td>
<td>Simple scalar variables, intraprocedural, flow-sensitive</td>
</tr>
<tr>
<td>Theory</td>
<td>Data-flow analysis (graphs &amp; solving fix-point equations)</td>
</tr>
</tbody>
</table>

2. Pointer alias analysis

<table>
<thead>
<tr>
<th>Goal</th>
<th>Used in program understanding, concrete type inference in OO programs (resolve target of method invocation, inline, and optimize)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>Pointers, interprocedural, flow-insensitive</td>
</tr>
<tr>
<td>Theory</td>
<td>Relations, Binary decision diagrams (BDD)</td>
</tr>
</tbody>
</table>

3. Parallelization and memory hierarchy optimization

<table>
<thead>
<tr>
<th>Goal</th>
<th>Parallelizes sequential programs (for multiprocessors)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Optimizes for the memory hierarchy</td>
</tr>
<tr>
<td>Scope</td>
<td>Arrays, loops</td>
</tr>
<tr>
<td>Theory</td>
<td>Linear algebra</td>
</tr>
</tbody>
</table>

4. Garbage collection (run-time system)
### Tentative Course Schedule

<table>
<thead>
<tr>
<th></th>
<th>Course Introduction</th>
<th>Date-flow analysis: introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Basic optimization</td>
<td>Data-flow analysis: theoretic foundation</td>
</tr>
<tr>
<td></td>
<td>Optimization: constant propagation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(joeq framework)</td>
<td>Optimization: redundancy elimination</td>
</tr>
<tr>
<td></td>
<td>Register allocation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instruction Scheduling</td>
<td>Non-numerical code scheduling</td>
</tr>
<tr>
<td></td>
<td>Software pipelining</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Loop Transformation</td>
<td>Parallelization</td>
</tr>
<tr>
<td></td>
<td>Loop transformations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data locality</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer alias analysis</td>
<td>Algorithm</td>
</tr>
<tr>
<td></td>
<td>BDDs in pointer analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Satisfiability modulo theories</td>
<td>Introduction</td>
</tr>
<tr>
<td></td>
<td>Garbage Collection</td>
<td>Basic concepts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Advanced concepts</td>
</tr>
</tbody>
</table>

### Course Emphasis

- **Methodology:** apply the methodology to other real life problems
  - Problem statement
    - Which problem to solve?
  - Theory and Algorithm
    - Theoretical frameworks
    - Algorithms
  - Experimentation: Hands-on experience

- **Compiler knowledge:**
  - Non-goal: how to build a complete optimizing compiler
  - Important algorithms
  - Exposure to new ideas
  - Background to learn existing techniques
Assignment by next class (no need to hand in)

- Think about how to build a compiler that converts the code on page 11 to page 12
  - (Read Chapter 9.1 for introduction of the optimizations)
- Example:
  Bubblesort program that sorts array A allocated in static storage

```c
for (i = n-2; i >= 0; i--) {
    for (j = 0; j <= i; j++) {
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
    }
}
```

Code Generated by the Front End

```c
i := n-2
S5: if i<=0 goto s1
j := 0
s4: if j>i goto s2
t1 = 4*j
t2 = &A
t3 = t2+t1
t4 = *t3 ;A[j]
t5 = j+1
t6 = 4*t5
t7 = &A
t8 = t7+t6
t9 = *t8 ;A[j+1]
t10 = 4*j
s3: j = j+1
t11 = &A
goto S4
t12 = t11+t10
S2: i = i-1
temp = *t12 ;temp=A[j]
goto s5
s1:
(t4=*t3 means read memory at address in t3 and write to t4: *t20=*t17: store value of t17 into memory at address in t20)
```
After Optimization

Result of applying:
- global common subexpression
- loop invariant code motion
- induction variable elimination
- dead-code elimination
to all the scalar and temp. variables

These traditional optimizations can make a big difference!

```plaintext
i = n-2
t27 = 4*i

t28 = &A
t29 = t27+t28
t30 = t28+4

s5: if t29 < t28 goto s1
t25 = t28
t26 = t30

s4: if t25 > t29 goto s2
t4 = *t25 ;A[j]
t9 = *t26 ;A[j+1]
if t4 <= t9 goto s3
temp = *t25 ;temp=A[j]
t17 = *t26 ;A[j+1]
*t25 = t17 ;A[j]=A[j+1]
*t26 = temp ;A[j+1]=temp

s3: t25 = t25+4
t26 = t26+4
goto s4

s2: t29 = t29-4
goto s5

s1:
```