Introduction to Modern Compilers (part of E0255)

Uday Kumar Reddy B

udayb@iisc.ac.in

Dept of CSA
Indian Institute of Science

Research in Programming and Compiler Technologies

- **Current:**
  - C, C++, Rust, Java, Python, MATLAB, R, ...
- **What will the new and disruptive programming technologies of the 21st century be?**

- **Current:**
  - C, C++, Rust, Java, Python, MATLAB, R, ...

- **What will the new and disruptive programming technologies of the 21st century be?**

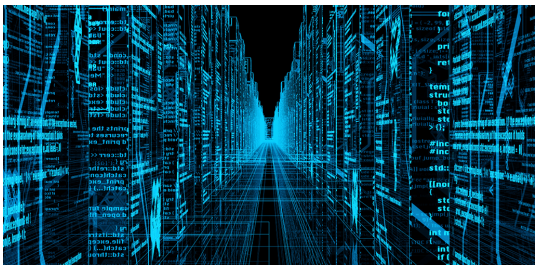
1. **What do programmers want?**
2. **How are architectures evolving?**
   - Multiple cores and many cores on a chip
   - GPUs, accelerators, and heterogeneous parallel architectures
   - Wider vector processing units
   - Deep memory hierarchies
   - Reduced precision

High-Performance Compilation: What do you want to program?

- Scientific and engineering simulations
  - Eg: Solving partial differential equations numerically
- Embedded vision (Eg: Autonomous/self-driving cars)
- Smartphones — HPC in data centers and cloud drives a number of smartphone technologies
- **Scientific and Engineering simulations**
- **Data Analytics**
- **Deep Learning**
- **Generative AI, LLMs**

Questions to Think About

- **What will the new programming technologies for the emerging domains be?**
  - **Current:** C, C++, Rust with OpenMP, MPI, CUDA, OpenCL
  - **Future: New languages, compilers, libraries, and DSLs**



Questions to Think About

- **What will the new programming technologies for AI be?**
  - PyTorch is dominant today; JAX is another high-level one. OpenAI Triton is mid-level.
  - **Just scratches the surface**

The Need for High Performance

- **More/Larger Data**
  - Instagram — 60 million photos / day
  - YouTube — 100 hours of video uploaded every minute
- **Need for a fast/real-time response in some domains**
- **More complex algorithms**
- **Science/Engineering simulations/modeling: Time to solution**

Programming Modern Hardware Effectively

- Compute speed: Eg.: 16 multiply-adds per cycle (AVX-512 unit for fp32)
- Synchronization (2 cores 0.25 μs, 8 cores 1.25 μs, 2x8 cores 1.54 μs)
- Memory bandwidth (10 GB/s per core, 500 GB/s per socket)
- High-Performance Programming and Compilation
  - Exploiting locality (caches, registers)
  - Exploit single core hardware well (vectorization, ...)
  - Multi-core parallelism
  - Reduce synchronization and communication as much as possible
- Good scaling without good single thread performance is a great waste of resources (power, equipment cost)

Programming Modern Hardware Effectively

- Compute speed: Eg.: 16 multiply-adds per cycle (AVX-512 unit for fp32)
- Synchronization (2 cores 0.25 μs, 8 cores 1.25 μs, 2x8 cores 1.54 μs)
- Memory bandwidth (10 GB/s per core, 500 GB/s per socket)
- High-Performance Programming and Compilation
  - Exploiting locality (caches, registers)
  - Exploit single core hardware well (vectorization, ...)
  - Multi-core parallelism
  - Reduce synchronization and communication as much as possible
- Good scaling without good single thread performance is a great waste of resources (power, equipment cost)

A Classification of Various Approaches

1. Manual low-level (C, C++) with parallel programming models (OpenMP, CUDA, MPI) with the best optimizing compilers
2. Library-based: C, C++, Python with libraries/packages: MKL, ScaLAPACK, CuBLAS, CuDNN, Cutlass, CuB
3. Mid-level: Triton, CuTile, Pallas
4. Ultra-high level languages and models including embedded DSLs: Tensorflow, PyTorch, JAX, R, MATLAB, Halide, Spiral

- General goal: Obtain productivity of the last class and the performance of the first

A Classification of Various Approaches

1. Manual low-level (C, C++) with parallel programming models (OpenMP, CUDA, MPI) with the best optimizing compilers
2. Library-based: C, C++, Python with libraries/packages: MKL, ScaLAPACK, CuBLAS, CuDNN, Cutlass, CuB
3. Mid-level: Triton, CuTile, Pallas
4. Ultra-high level languages and models including embedded DSLs: Tensorflow, PyTorch, JAX, R, MATLAB, Halide, Spiral

- General goal: Obtain productivity of the last class and the performance of the first

A Classification of Various Approaches

1. Manual low-level (C, C++) with parallel programming models (OpenMP, CUDA, MPI) with the best optimizing compilers
2. Library-based: C, C++, Python with libraries/packages: MKL, ScaLAPACK, CuBLAS, CuDNN, Cutlass, CuB
3. Mid-level: Triton, CuTile, Pallas
4. Ultra-high level languages and models including embedded DSLs: Tensorflow, PyTorch, JAX, R, MATLAB, Halide, Spiral

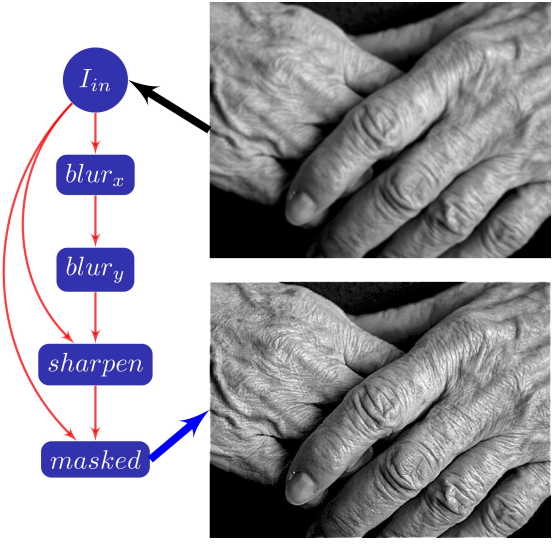
- **General goal**: Obtain productivity of the last class and the performance of the first

A Classification of Various Approaches

1. Manual low-level (C, C++) with parallel programming models (OpenMP, CUDA, MPI) with the best optimizing compilers
2. Library-based: C, C++, Python with libraries/packages: MKL, ScaLAPACK, CuBLAS, CuDNN, Cutlass, CuB
3. Mid-level: Triton, CuTile, Pallas
4. Ultra-high level languages and models including embedded DSLs: Tensorflow, PyTorch, JAX, R, MATLAB, Halide, Spiral

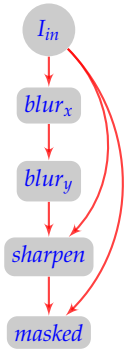
- **General goal**: Obtain productivity of the last class and the performance of the first

EXAMPLE: UNSHARP MASK – AN IMAGE PROCESSING PIPELINE



UNSHARP MASK: COMPUTATION

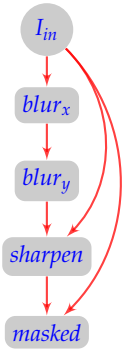
```
for (i = 0; i <= 2; i++)
  for (j = 2; j <= (R + 1); j++)
    for (k = 0; (k <= (C + 3)); k++)
      blurx[i][j-2][k] = img[i][j-2][k]*0.0625f + img[i][j-1][k]*0.25f
       + img[i][j][k]*0.375f + img[i][j+1][k]*0.25f + img[i][j+2][k]*0.0625f;

for (i = 0; (i <= 2); i++)
  for (j = 2; (j <= (R + 1)); j++)
    for (k = 2; (k <= (C + 1)); k++)
      blury[i][j][k-2] = blurx[i][j-2][k-2]*0.0625f + blurx[i][j-2][k-1]*0.25f
            + blurx[i][j-2][k]*0.375f + blurx[i][j-2][k+1]*0.25f + blurx[i][j-2][k+2]*0.0625f;

for (i = 0; (i <= 2); i++)
  for (j = 2; (j <= (R + 1)); j++)
    for (k = 2; (k <= (C + 1)); k++)
      sharpen[i][j][k-2] = img[i][j][k]*(1 + weight) + blury[i][j-2][k-2]*(-weight);

for (i = 0; i <= 2; i++)
  for (j = 2; j <= R + 1; j++)
    for (k = 2; k <= C + 1; k++) {
      _ct0 = img[i][j][k];
      _ct1 = sharpen[i][j-2][k-2];
      _ct2 = (std::abs((img[i][j][k] - blury[i][j-2][k-2])) < threshold)? _ct0: _ct1;
      mask[i][j-2][k-2] = _ct2;
    }
```



A sequential version in C: **18.6 ms** / frame
(using GCC with opts, quad-core Nehalem, 720p video)

UNSHARP MASK - A NAIVE OPENMP VERSION

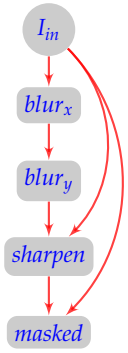
```
for (i = 0; i <= 2; i++)
#pragma omp parallel for
  for (j = 2; j <= (R + 1); j++)
#pragma ivdep
    for (k = 0; k <= C + 3; k++)
      blurx[i][j-2][k] = img[i][j-2][k]*0.0625f + img[i][j-1][k]*0.25f
       + img[i][j][k]*0.375f + img[i][j+1][k]*0.25f + img[i][j+2][k]*0.0625f;

for (i = 0; i <= 2; i++)
#pragma omp parallel for
  for (j = 2; j <= R + 1; j++)
#pragma ivdep
    for (k = 2; k <= C + 1; k++)
      blury[i][j][k-2] = blurx[i][j-2][k-2]*0.0625f + blurx[i][j-2][k-1]*0.25f
            + blurx[i][j-2][k]*0.375f + blurx[i][j-2][k+1]*0.25f + blurx[i][j-2][k+2]*0.0625f;

for (i = 0; i <= 2; i++)
#pragma omp parallel for
  for (j = 2; j <= R + 1; j++)
#pragma ivdep
    for (k = 2; k <= C + 1; k++)
      sharpen[i][j][k-2] = img[i][j][k]*(1 + weight) + blury[i][j-2][k-2]*(-weight);

for (i = 0; i <= 2; i++)
#pragma omp parallel for private(_ct0,_ct1,_ct2)
  for (j = 2; j <= R + 1; j++)
#pragma ivdep
    for (k = 2; k <= C + 1; k++) {
      _ct0 = img[i][j][k];
      _ct1 = sharpen[i][j-2][k-2];
      _ct2 = (std::abs((img[i][j][k] - blury[i][j-2][k-2])) < threshold)? _ct0: _ct1;
      mask[i][j-2][k-2] = _ct2;
    }
```



20.2 ms / frame on 1 thread, **18.02 ms** / frame on 4 threads

UNSHARP MASK - A BETTER OPENMP VERSION

```
#pragma omp parallel for
for (j = 2; j <= (R + 1); j++)
  for (i = 0; i <= 2; i++)
#pragma ivdep
    for (k = 0; (k <= (C + 3)); k++)
      blurx[i][j-2][k] = img[i][j-2][k]*0.0625f + img[i][j-1][k]*0.25f
       + img[i][j][k]*0.375f + img[i][j+1][k]*0.25f + img[i][j+2][k]*0.0625f;

#pragma omp parallel for
for (j = 2; (j <= (R + 1)); j++)
  for (i = 0; i <= 2; i++)
#pragma ivdep
    for (k = 2; (k <= (C + 1)); k++)
      blury[i][j][k-2] = blurx[i][j-2][k-2]*0.0625f + blurx[i][j-2][k-1]*0.25f
            + blurx[i][j-2][k]*0.375f + blurx[i][j-2][k+1]*0.25f + blurx[i][j-2][k+2]*0.0625f;

#pragma omp parallel for
for (j = 2; (j <= (R + 1)); j++)
  for (i = 0; i <= 2; i++)
#pragma ivdep
    for (k = 2; (k <= (C + 1)); k++)
      sharpen[i][j][k-2] = img[i][j][k]*(1 + weight) + blury[i][j-2][k-2]*(-weight);

#pragma omp parallel for private(_ct0,_ct1,_ct2)
for (j = 2; j <= R + 1; j++)
  for (i = 0; i <= 2; i++)
#pragma ivdep
    for (k = 2; k <= C + 1; k++) {
      _ct0 = img[i][j][k];
      _ct1 = sharpen[i][j-2][k-2];
      _ct2 = (std::abs((img[i][j][k] - blury[i][j-2][k-2])) < threshold)? _ct0: _ct1;
      mask[i][j-2][k-2] = _ct2;
    }
```



18.6 ms / frame on 1 thread, **15.03 ms** / frame on 4 threads

Optimizing Unsharp Mask

1. Write with OpenCV library (with Python bindings)

```
@jit("float32[::](uint8[::],␣int64)", cache = True, nogil = True)
def unsharp_cv(frame, lib_func):
  frame_f = np.float32(frame) / 255.0
  res = frame_f
  kernelx = np.array([1, 4, 6, 4, 1], np.float32) / 16
  kernely = np.array([[1], [4], [6], [4], [1]], np.float32) / 16
  blury = sepFilter2D(frame_f, -1, kernelx, kernely)
  sharpen = addWeighted(frame_f, (1 + weight), blury, (-weight), 0)
  th, choose = threshold(absdiff(frame_f, blury), thresh, 1, THRESH_BINARY)
  choose = choose.astype(bool)
  np.copyto(res, sharpen, 'same_kind', choose)
  return res
```

   Performance: **35.9 ms** / frame

2. Write in a dynamic language like Python and use a JIT (Numba) — performance: **79 ms / frame**
3. A naive C version parallelized with OpenMP: **18.02 ms** / frame
4. A version with sophisticated optimizations (fusion + overlapped tiling): **8.97 ms** / frame (in this course, we will study how to get to this, and build compilers/code generators that can achieve this automatically)

- **Video demo**

Optimizing Unsharp Mask

1. Write with OpenCV library (with Python bindings)

```
@jit("float32[::](uint8[::],␣int64)", cache = True, nogil = True)
def unsharp_cv(frame, lib_func):
  frame_f = np.float32(frame) / 255.0
  res = frame_f
  kernelx = np.array([1, 4, 6, 4, 1], np.float32) / 16
  kernely = np.array([[1], [4], [6], [4], [1]], np.float32) / 16
  blury = sepFilter2D(frame_f, -1, kernelx, kernely)
  sharpen = addWeighted(frame_f, (1 + weight), blury, (-weight), 0)
  th, choose = threshold(absdiff(frame_f, blury), thresh, 1, THRESH_BINARY)
  choose = choose.astype(bool)
  np.copyto(res, sharpen, 'same_kind', choose)
  return res
```

   Performance: **35.9 ms** / frame

2. Write in a dynamic language like Python and use a JIT (Numba) — performance: **79 ms / frame**
3. A naive C version parallelized with OpenMP: **18.02 ms** / frame
4. A version with sophisticated optimizations (fusion + overlapped tiling): **8.97 ms** / frame (in this course, we will study how to get to this, and build compilers/code generators that can achieve this automatically)

- **Video demo**

Optimizing Unsharp Mask

1. Write with OpenCV library (with Python bindings)

```
@jit("float32[::](uint8[::],␣int64)", cache = True, nogil = True)
def unsharp_cv(frame, lib_func):
  frame_f = np.float32(frame) / 255.0
  res = frame_f
  kernelx = np.array([1, 4, 6, 4, 1], np.float32) / 16
  kernely = np.array([[1], [4], [6], [4], [1]], np.float32) / 16
  blury = sepFilter2D(frame_f, -1, kernelx, kernely)
  sharpen = addWeighted(frame_f, (1 + weight), blury, (-weight), 0)
  th, choose = threshold(absdiff(frame_f, blury), thresh, 1, THRESH_BINARY)
  choose = choose.astype(bool)
  np.copyto(res, sharpen, 'same_kind', choose)
  return res
```

   Performance: **35.9 ms** / frame

2. Write in a dynamic language like Python and use a JIT (Numba) — performance: **79 ms / frame**
3. A naive C version parallelized with OpenMP: **18.02 ms** / frame
4. A version with sophisticated optimizations (fusion + overlapped tiling): **8.97 ms** / frame (in this course, we will study how to get to this, and build compilers/code generators that can achieve this automatically)

- **Video demo**

Optimizing Unsharp Mask

1. Write with OpenCV library (with Python bindings)

```
@jit("float32[::](uint8[::],␣int64)", cache = True, nogil = True)
def unsharp_cv(frame, lib_func):
  frame_f = np.float32(frame) / 255.0
  res = frame_f
  kernelx = np.array([1, 4, 6, 4, 1], np.float32) / 16
  kernely = np.array([[1], [4], [6], [4], [1]], np.float32) / 16
  blury = sepFilter2D(frame_f, -1, kernelx, kernely)
  sharpen = addWeighted(frame_f, (1 + weight), blury, (-weight), 0)
  th, choose = threshold(absdiff(frame_f, blury), thresh, 1, THRESH_BINARY)
  choose = choose.astype(bool)
  np.copyto(res, sharpen, 'same_kind', choose)
  return res
```

   Performance: **35.9 ms** / frame

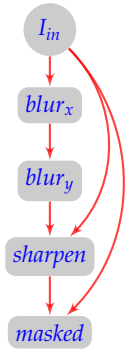
2. Write in a dynamic language like Python and use a JIT (Numba) — performance: **79 ms / frame**
3. A naive C version parallelized with OpenMP: **18.02 ms** / frame
4. A version with sophisticated optimizations (fusion + overlapped tiling): **8.97 ms** / frame (in this course, we will study how to get to this, and build compilers/code generators that can achieve this automatically)

- **Video demo**

UNSHARP MASK - A HIGHLY OPTIMIZED VERSION

Note: *Code below is indicative and not meant for reading. Zoom into soft copy or browse source code repo listed in references.*

```
#pragma omp parallel for schedule(static)
for (int _T_11 = 0; (_T_11 <= ((R + 1) / 32)); _T_11 = (_T_11 + 1))
{
  int _ct0 = (((R + 1) < ((32 * _T_11) + 31))? (R + 1): ((32 * _T_11) + 31));
  int _ct1 = ((2 > (32 * _T_11))? 2: (32 * _T_11));
  int _ct4 = (((R + 1) < ((32 * _T_11) + 31))? (R + 1): ((32 * _T_11) + 31));
  int _ct5 = ((2 > (32 * _T_11))? 2: (32 * _T_11));
  int _ct8 = (((R + 1) < ((32 * _T_11) + 31))? (R + 1): ((32 * _T_11) + 31));
  int _ct9 = ((2 > (32 * _T_11))? 2: (32 * _T_11));
  int _ct12 = (((R + 1) < ((32 * _T_11) + 31))? (R + 1): ((32 * _T_11) + 31));
  int _ct13 = ((2 > (32 * _T_11))? 2: (32 * _T_11));
  for (int _T_12 = -1; (_T_12 <= (((C + 3) / 256))); _T_12 = (_T_12 + 1))
  {
    int _ct2 = (((C + 3) < ((256 * _T_12) + 261))? (C + 3): ((256 * _T_12) + 261));
    int _ct3 = (((0 > (256 * _T_12))? 0: (256 * _T_12));
    int _ct6 = (((C + 1) < ((256 * _T_12) + 260))? (C + 1): ((256 * _T_12) + 260));
    int _ct7 = ((2 > ((256 * _T_12) + 1))? 2: ((256 * _T_12) + 1));
    int _ct10 = (((C + 1) < ((256 * _T_12) + 258))? (C + 1): ((256 * _T_12) + 259));
    int _ct11 = ((2 > ((256 * _T_12) + 2))? 2: ((256 * _T_12) + 2));
    int _ct14 = (((C + 1) < ((256 * _T_12) + 258))? (C + 1): ((256 * _T_12) + 258));
    int _ct15 = ((2 > ((256 * _T_12) + 3))? 2: ((256 * _T_12) + 3));
    for (int _i0 = 0; (_i0 <= 2); _i0 = (_i0 + 1))
    {
      for (int _i1 = _ct1; (_i1 <= _ct0); _i1 = (_i1 + 1))
      {
        #pragma ivdep
        for (int _i2 = _ct3; (_i2 <= _ct2); _i2 = (_i2 + 1))
        {
          _blurx[_i0][(_i2 - _T_11)][(_i1 - _t1)][((-256 * _T_12) + _i2)] = ((((img[((_i0 + (R + 4) + (C + 4))) + ((-2 + _i1) * (C + 4))) + _i2)] * 0.06255f) + (img[(((_i0 + ((R + 4) * (C + 4))) + ((-1 + _i1) * (C + 4))) + _i2)] * 0.25f)) + (img[(((_i0 + ((R + 4) * (C + 4))) + _i1) * (C + 4)) + _i2)] * 0.38751f)) + (img[(((_i0 + ((R + 4) * (C + 4))) + ((1 + _i1) * (C + 4))) + _i2)] * 0.25f)) + (img[(((_i0 + ((R + 4) * (C + 4))) + ((2 + _i1) * (C + 4))) + _i2)] * 0.06255f));
        }
      }
    }
    for (int _i0 = 0; (_i0 <= 2); _i0 = (_i0 + 1))
    {
      for (int _i1 = _ct5; (_i1 <= _ct4); _i1 = (_i1 + 1))
      {
        #pragma ivdep
        for (int _i2 = _ct7; (_i2 <= _ct6); _i2 = (_i2 + 1))
        {
          _blury[_i0][(_i2 - _T_11)][(_i1 - _t1)][((-256 * _T_12) + _i2)] = ((((_blurx[_i0][((-2 * _T_11) + _i1)][(-2 + ((-256 * _T_12) + _i2))] * 0.06255f) + (_blurx[_i0][((-1 * _T_11) + _i1)][(-1 + ((-256 * _T_12) + _i2))] * 0.25f)) + (_blurx[_i0][((-2 * _T_11) + _i1)][((-256 * _T_12) + _i2)] * 0.38751f)) + (_blurx[_i0][((-1 * _T_11) + _i1)][(1 + ((-256 * _T_12) + _i2))] * 0.25f)) + (_blurx[_i0][((-2 * _T_11) + _i1)][(2 + ((-256 * _T_12) + _i2))] * 0.06255f));
        }
      }
    }
    for (int _i0 = 0; (_i0 <= 2); _i0 = (_i0 + 1))
    {
      for (int _i1 = _ct9; (_i1 <= _ct8); _i1 = (_i1 + 1))
      {
        #pragma ivdep
        for (int _i2 = _ct11; (_i2 <= _ct10); _i2 = (_i2 + 1))
        {
          sharpen[_i0][((-32 + _T_11) + _i1)][((-256 + _T_12) + _i2)] = (((img[(((_i0 + ((R + 4) + (C + 4))) + (_i1 + (C + 4))) + _i2)] + (1 + weight)) + (blury[_i0][((-32 + _T_11) + _i1)][((-256 + _T_12) + _i2)] * (-weight)));
        }
      }
    }
    for (int _i0 = 0; (_i0 <= 2); _i0 = (_i0 + 1))
    {
      for (int _i1 = _ct13; (_i1 <= _ct12); _i1 = (_i1 + 1))
      {
        #pragma ivdep
        for (int _i2 = _ct15; (_i2 <= _ct14); _i2 = (_i2 + 1))
        {
          float _ct16 = img[(((_i0 + ((R + 4) * (C + 4))) + (_i1 * (C + 4))) + _i2)];
          float _ct17 = sharpen[_i0][((-32 + _T_11) + _i1)][((-256 + _T_12) + _i2)];
          float _ct18 = (((fabs(abs((img[(((_i0 + ((R + 4) * (C + 4))) + (_i1 * (C + 4))) + _i2)] - blury[_i0][((-32 + _T_11) + _i1)][((-256 + _T_12) + _i2)])) < threshold))? _ct16: _ct17);
          mask_fig[(((_i1-2) + (3 * C)) + ((_i2 - 2) + 3))] = (_ct0)? _ct18;
        }
      }
    }
  }
}
```



15.5 ms / frame on 1 threads, **8.97 ms** / frame on 4 threads

Domain-Specific Languages (DSL)

- **The example motivates a domain-specific language + compiler approach**
- **High-performance domain-specific language + compiler**: productivity similar to ultra high-level or high-level but performance similar to manual or even better!

Domain-Specific Languages (DSL)

- **The example motivates a domain-specific language + compiler approach**
- **High-performance domain-specific language + compiler**: productivity similar to ultra high-level or high-level but performance similar to manual or even better!

Domain-Specific Languages (DSL)

DSLs

- Exploit domain information to improve programmability, performance, and portability
- Expose greater information to the compiler and programmer specifies less
- abstract away many things from programmers (parallelism, memory)

DSL compilers

- can "see" **across** routines – allow whole program optimization
- generate optimized code for multiple targets
- Programmers say **what** to execute and not **how** to execute

Domain-Specific Languages (DSL)

DSLs

- Exploit domain information to improve programmability, performance, and portability
- Expose greater information to the compiler and programmer specifies less
- abstract away many things from programmers (parallelism, memory)

DSL compilers

- can "see" **across** routines – allow whole program optimization
- generate optimized code for multiple targets
- Programmers say **what** to execute and not **how** to execute

Big Picture: Role of Compilers

General-Purpose

- Improve existing **general-purpose** compilers (for C, C++, Python, ...)
- Programmers say a **LOT**
- LLVM/Polly, GCC/Graphite

Domain-Specific

- Build new **domain-specific languages and compilers**
- Programmers say **WHAT** they execute and not **HOW** they execute
- SPIRAL, Halide, Tensorflow, Pytorch, ...

Big Picture: Role of Compilers

General-Purpose

- Improve existing **general-purpose** compilers (for C, C++, Python, ...)
- Programmers say a **LOT**
- LLVM/Polly, GCC/Graphite
- Limited improvements, not everything is possible
- Broad impact

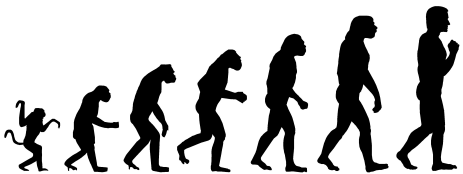
Domain-Specific

- Build new **domain-specific languages and compilers**
- Programmers say **WHAT** they execute and not **HOW** they execute
- SPIRAL, Halide, Tensorflow, Pytorch, ...
- Dramatic speedups, Automatic parallelization
- Narrower impact and adoption

Big Picture: Role of Compilers

EVOLUTIONARY approach

- Improve existing **general-purpose** compilers (for C, C++, Python, ...)
- Programmers say a **LOT**
- LLVM/Polly, GCC/Graphite



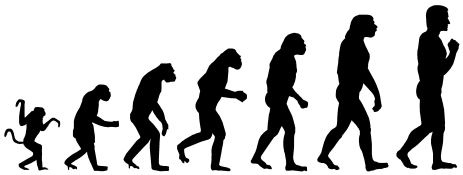
REVOLUTIONARY approach

- Build new **domain-specific languages and compilers**
- Programmers say **WHAT** they execute and not **HOW** they execute
- SPIRAL, Halide, Tensorflow, Pytorch, ...

BIG PICTURE: ROLE OF COMPILERS

EVOLUTIONARY approach

- Improve existing **general-purpose** compilers (for C, C++, Python, ...)
- Programmers say a **LOT**
- LLVM/Polly, GCC/Graphite



REVOLUTIONARY approach

- Build new **domain-specific languages and compilers**
- Programmers say **WHAT** they execute and not **HOW** they execute
- SPIRAL, Halide, Tensorflow, Pytorch, ...

- **Both approaches share infrastructure**
- **Important to pursue both**

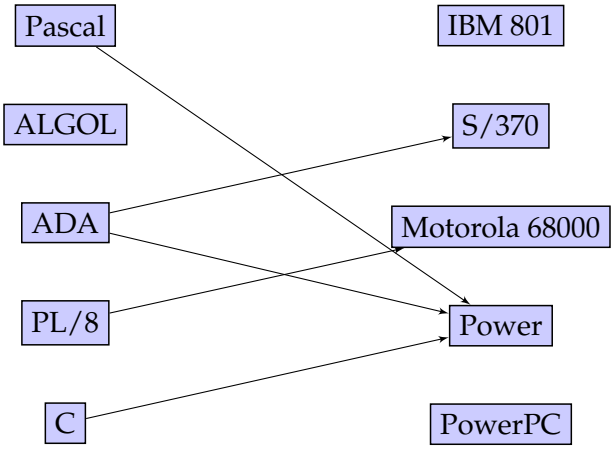
Outline

- Compilers for AI

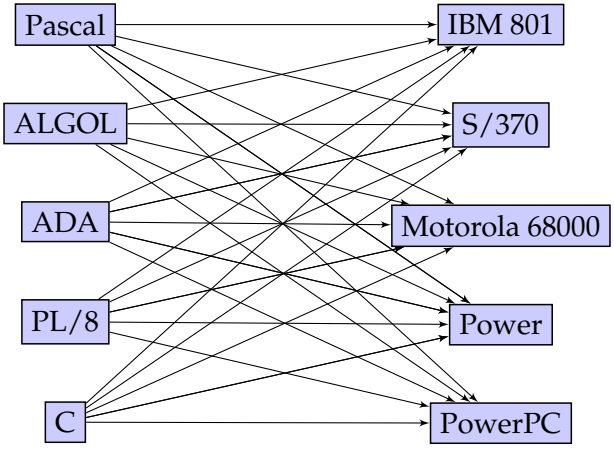
Compilers for AI

- **Compilers** are language translators: they translate programming languages to instructions hardware can execute
- One of the pillars of Computer Systems

Compilers - The Early Days

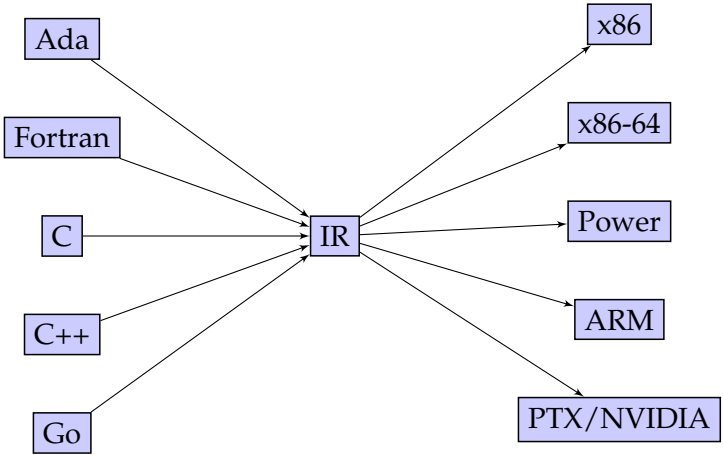


Compilers - The Early Days



- M **languages,** N **targets** $\Rightarrow M * N$ **compilers! Not scalable!**

COMPILERS EVOLUTION - $M + N$



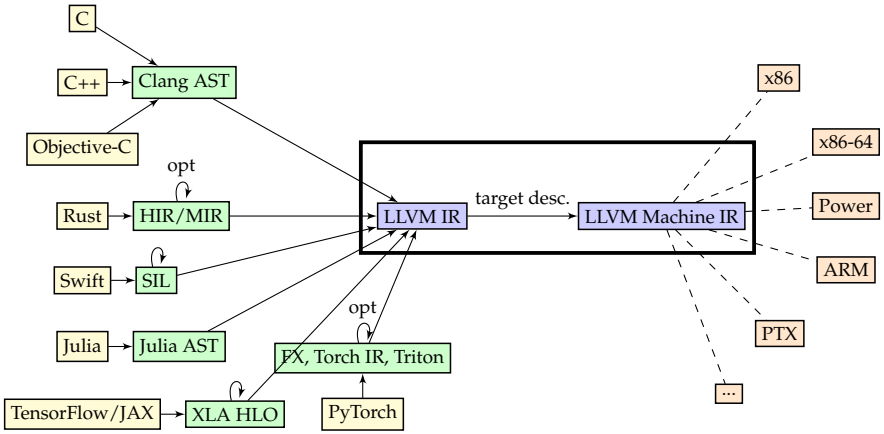
- **With an common IR, we have $M + N + 1$ compilers!**

What does an IR look like?

- A representation convenient to analyze and transform
- Round-trippable form that you can parse and print
- Low-level IRs are three-address code-like
- IRs have used expressions trees, 3-address code, graphs.
- Static Single Assignment: a property of IRs that makes it convenient; most IRs now use SSA

```
define void @foo(ptr nocapture %a) {
entry:
  br label %for.body

for.body:   ; preds = %for.body, %entry
  %indvars.iv = phi i64 [ 0, %entry ], [ %indvars.iv.next, %for.body ]
  %arrayidx = getelementptr inbounds i32, ptr %a, i64 %indvars.iv
  %0 = load i32, ptr %arrayidx, align 4
  %1 = add i32 %0, 2
  %inc = add nsw i32 %0, 1
  store i32 %inc, ptr %arrayidx, align 4
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond = icmp eq i64 %indvars.iv.next, 64
  br i1 %exitcond, label %for.end, label %for.body

for.end:        ; preds = %for.body
  ret void
}
```

Modern Compilers - LLVM IR-based



- LLVM: modular, reusable, open-source, but too low-level, not extensible for higher-order languages.

COMPILERS FOR AI

AI programming frameworks

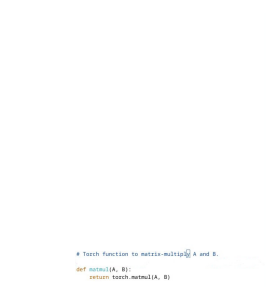
. . . ?

Compiler infrastructure?

Explosion of AI chips and accelerators

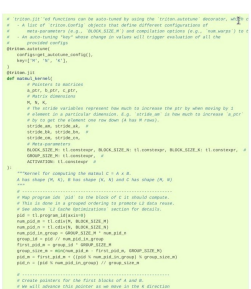
- Space in between is ruled by hand-written libraries. Not scalable.
- The right tools weren't available until recently.

How do you program AI hardware?



```
# Torch function to matrix-multiply A and B.

def matmul(A, B):
    return torch.matmul(A, B)
```

PyTorch-



```
# `triton.jit`'ed functions can be auto-tuned by using the `triton.autotune` decorator, which
#   - A list of `triton.Config` objects that define different configurations of
#     meta-parameters (e.g., `BLOCK_SIZE_M`) and compilation options (e.g., `num_warps`) to
#   - An auto-tuning *key* whose change in values will trigger evaluation of all the
#     provided configs
@triton.autotune(
    configs=get_autotune_config(),
    key=['M', 'N', 'K'],
)
@triton.jit
def matmul_kernel(
        # Pointers to matrices
        a_ptr, b_ptr, c_ptr,
        # Matrix dimensions
        M, N, K,
        # The stride variables represent how much to increase the ptr by when moving by 1
        # element in a particular dimension. E.g. `stride_am` is how much to increase `a_ptr`
        # by to get the element one row down (A has M rows).
        stride_am, stride_ak,
        stride_bk, stride_bn,
        stride_cm, stride_cn,
        # Meta-parameters
        BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,
        GROUP_SIZE_M: tl.constexpr,
        ACTIVATION: tl.constexpr
):
    """Kernel for computing the matmul C = A x B.
    A has shape (M, K), B has shape (K, N) and C has shape (M, N)
    """
    # -----------------------------------------------------------
    # Map program ids `pid` to the block of C it should compute.
    # This is done in a grouped ordering to promote L2 data reuse.
    # See above `L2 Cache Optimizations` section for details.
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
    pid_m = first_pid_m + (pid % num_pid_in_group) % group_size_m
    pid_n = (pid % num_pid_in_group) // group_size_m

    # ----------------------------------------------------------
    # Create pointers for the first blocks of A and B.
    # We will advance this pointer as we move in the K direction
    # and accumulate
    # `a_ptrs` is a block of [BLOCK_SIZE_M, BLOCK_SIZE_K] pointers
    # `b_ptrs` is a block of [BLOCK_SIZE_K, BLOCK_SIZE_N] pointers
    # See above `Pointer arithmetic` section for details
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)

    # -----------------------------------------------------------
    # Iterate to compute a block of the C matrix.
    # We accumulate into a `[BLOCK_SIZE_M, BLOCK_SIZE_N]` block
    # of fp32 values for higher accuracy.
    # `accumulator` will be converted back to fp16 after the loop.
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        # Load the next block of A and B, generate a mask by checking the K dimension.
        # If it is out of bounds, set it to 0.
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        # We accumulate along the K dimension.
        accumulator = tl.dot(a, b, accumulator)
        # Advance the ptrs to the next K block.
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    # You can fuse arbitrary activation functions here
    # while the accumulator is still in FP32!
    if ACTIVATION == "leaky_relu":
        accumulator = leaky_relu(accumulator)
    c = accumulator.to(tl.float16)

    # -----------------------------------------------------------
    # Write back the block of the output matrix C with masks.
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

Triton

CUDA

- High, mid, and low-level abstractions
- High: PyTorch, JAX, ...
- Mid: OpenAI Triton, cuTile
- Low: CUDA, C/C++, CUTLASS, ...
- All three approaches need/use compilers in different ways
- They also share/rest on the same underlying infrastructure
- Eg: Triton, MLIR, LLVM, PTX

How do you program AI hardware?

- High, mid, and low-level abstractions
- High: PyTorch, JAX, ...
- Mid: OpenAI Triton, cuTile
- Low: CUDA, C/C++, CUTLASS, ...
- All three approaches need/use compilers in different ways
- They also share/rest on the same underlying infrastructure
- Eg: Triton, MLIR, LLVM, PTX

Python-based programming frameworks: Today

- Where does performance in Python-based frameworks come from?
- Largely from libraries written in C, C++, CUDA, and even assembly
- Compilers exist: XLA, TorchInductor, TensorRT
  - Limited in many ways: "semi-compilers", fragmented infra, performance
  - Still evolving

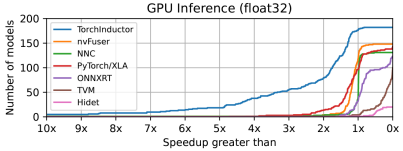
```python
class SelfAttentionLayer(nn.Module):
    def __init__(self, feature_size):
        super(SelfAttentionLayer, self).__init__()
        self.feature_size = feature_size

        # Linear transformations for Q, K, V from the same source.
        self.key = nn.Linear(feature_size, feature_size)
        self.query = nn.Linear(feature_size, feature_size)
        self.value = nn.Linear(feature_size, feature_size)

    def forward(self, x, mask=None):
        # Apply linear transformations.
        keys = self.key(x)
        queries = self.query(x)
        values = self.value(x)

        # Scaled dot-product attention.
        scores = torch.matmul(queries, keys.transpose(-2, -1))
             / torch.sqrt(torch.tensor(self.feature_size, dtype=torch.float32))

        # Apply mask (if provided).
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # Apply softmax.
        attention_weights = F.softmax(scores, dim=-1)

        # Multiply weights with values.
        output = torch.matmul(attention_weights, values)

        return output, attention_weights
```

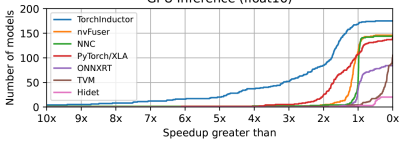
Opening multi-level doors to programming AI hardware

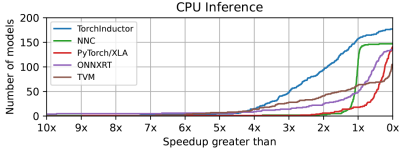
1. High-level Python-based programming frameworks (e.g. PyTorch, JAX),
2. Compiler support for (1) that could be turned off/on (e.g. torch.compile),
3. Mid/low-level programming support (e.g., CUDA, CUTLASS, CuTile, Triton)
4. Low-level MLIR dialects that expose their hardware intrinsics/virtual ISA on top of which both (2) compilers and (3) low-level frameworks rest,
5. Ability to use inline virtual ISA.

Compiler Auto-Parallelization is already here!

- Recent PyTorch 2 ASPLOS paper
  PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation, Ansel et al. (Meta), ASPLOS 2024.







How is hardware evolving? (1/2)

- Multiple cores (early 2000s)
- Wider SIMD (early 2000s)
- Many cores (late 2000s)
- Heterogeneity (2000s/2010s)
- Tensor/matmul cores (mid 2010s)
- Low-precision compute instructions (late 2010s/2020s)

From 2000s to now

How is hardware evolving? (2/2)

Example: NVIDIA H100 chip

- 80 GB of GPU DRAM
- 3.35 TB/s of memory bandwidth (HBM3).
- 990 TFLOPS for fp16 tensor operations, 1.98 PFLOPS for int8.
- 50 MB of L2 cache.

How are programming frameworks evolving?

- Programmer productivity
- Write less and do more
- Hardware usability
- Deliver performance
- Deliver portability

AI programming frameworks

COMPILERS FOR AI

AI programming frameworks

. . . ?

Compiler infrastructure?

Explosion of AI chips and accelerators

- MLIR infrastructure: open-sourced by Google in 2019

COMPILERS FOR AI

AI programming frameworks

. . . ?

Explosion of AI chips and accelerators

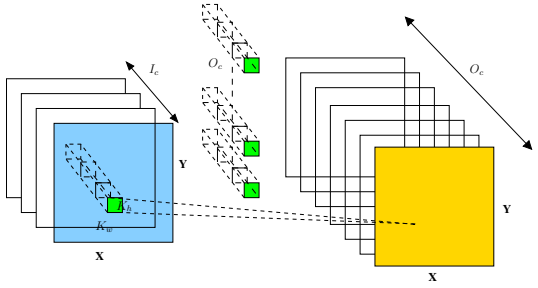
- MLIR infrastructure: open-sourced by Google in 2019

MLIR

- ML in MLIR: Multi-level
- Characteristics
  - Loops and multi-dimensional arrays (tensors) had to be first class citizens
  - Had to be extensible (types, operations, attributes)
  - Had to enable building both general-purpose and domain-specific compilers and even more.
  - Had to be open-source with a permissive license

Multi-dimensionality everywhere: CNN Convolution

```
for (n = 0; n < N; n++) // Samples in a batch.
  for (o = 0; o < Oc; o++) // Output feature channels.
    for (i = 0; i2 < Ic; i++) // Input feature channels.
      for (y = 0; i3 < Y; i3++) // Layer height.
        for (x = 0; i4 < X; i4++) // Layer width.
          for (kh = 0; i5 < Kh; i5++) // Convolution kernel height.
            for (kw = 0; i6 < Kw; i6++) // Convolution kernel width.
              output[n, o, y, x] += input[n, i, y+kh, x+kw] * weights[o, i, kh, kw];
```

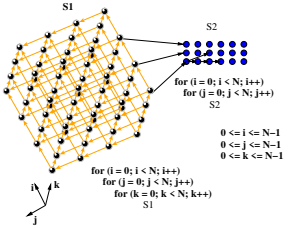


MLIR: Multi-level Intermediate Representation

1. Ops (general purpose to domain specific) on tensor types / graph form

```
%patches = "tf.reshape"(%patches, %minus_one, %minor_dim_size)
                : (tensor<? x ? x ? x ? x f32>, index, index) -> tensor<? x ? x f32>
%mat_out = "tf.matmul"(%patches_flat, %patches_flat){transpose_a : true}
                : (tensor<? x ? x f32>, tensor<? x ? x f32>) -> tensor<? x ? x f32>
%vec_out = "tf.reduce_sum"(%patches_flat) {axis: 0} : (tensor<? x ? x f32>) -> tensor<? x f32>
```

2. Loop-level / mid-level form




```
affine.for %i = 0 to 8 step 4 {
  affine.for %j = 0 to 8 step 4 {
    affine.for %k = 0 to 8 step 4 {
      affine.for %ii = #map0(%i) to #map1(%i) {
        affine.for %jj = #map0(%j) to #map1(%j) {
          affine.for %kk = #map0(%k) to #map1(%k) {
            %5 = affine.load %arg0[%ii, %kk] : memref<8 x 8 x vector<64 x f32>>
            %6 = affine.load %arg1[%kk, %jj] : memref<8 x 8 x vector<64 x f32>>
            %7 = affine.load %arg2[%ii, %jj] : memref<8 x 8 x vector<64 x f32>>
            %8 = arith.mulf %5, %6 : vector<64xf32>
            %9 = arith.addf %7, %8 : vector<64xf32>
            affine.store %9, %arg2[%ii, %jj] : memref<8 x 8 x vector<64xf32>>
          }
        }
      }
    }
  }
}
```

3. Low-level form: closer to hardware

```
%v1 = memref.load %a[%i2, %i3] : memref<256 x 64 x vector<16 x f32>>
%v2 = memref.load %b[%i2, %i3] : memref<256 x 64 x vector<16 x f32>>
%v3 = addf %v1, %v2 : vector<16 x f32>
memref.store %v3, %d[%i2, %i3] : memref<256 x 64 x vector<16 x f32>>
```

Modern Compiler Topics in this course

1. Foundations: SSA, Dominance, Basic concepts for control flow analysis and data flow analysis
2. Compiler optimizations for parallelism and locality
3. Affine abstraction/Polyhedral framework (only the basics)
4. MLIR
5. Practice: Building compilers using MLIR
6. Practice: Building compilers and optimizers for AI frameworks (basic overview, pointers)