E0255 Compiler Design

Uday Kumar Reddy Bondhugula

udayb@iisc.ac.in

Dept of CSA
Indian Institute of Science

Outline

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?**

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?**

Research in Programming and Compiler Technologies

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

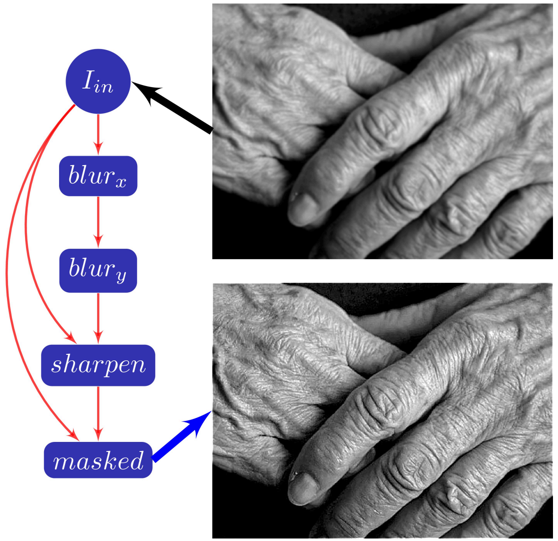
- **General goal**: Obtain productivity of the last class and the performance of the first

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

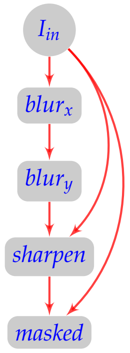
```
for (i = 0; i <= 2; i++)
  for (j = 2; j <= (R + 1); j++)
    for (k = 0; (k <= (C + 3)); k++)
      blurx[i][j-2][k] = img[i][j-2][k]*0.0625f + img[i][j-1][k]*0.25f
       + img[i][j][k]*0.375f + img[i][j+1][k]*0.25f + img[i][j+2][k]*0.0625f;

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;
    }
```

$$\begin{tikzcd}
I_{in} \arrow[d] \arrow[ddd, bend left] \arrow[dddd, bend left] \\
blur_x \arrow[d] \\
blur_y \arrow[d] \\
sharpen \arrow[d] \\
masked
\end{tikzcd}$$

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;
    }
```

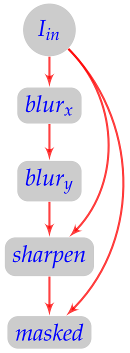

$$\begin{tikzcd} I_{in} \arrow[d] \arrow[ddd, bend left] \arrow[dddd, bend left] \\ blur_x \arrow[d] \\ blur_y \arrow[d] \\ sharpen \arrow[d] \\ masked \end{tikzcd}$$

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


$$\begin{tikzcd}
I_{in} \arrow[d] \arrow[ddd, bend left] \arrow[dddd, bend left] \\
blur_x \arrow[d] \\
blur_y \arrow[d] \\
sharpen \arrow[d] \\
masked
\end{tikzcd}$$

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

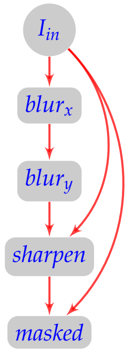
2. Write in a dynamic language like Python and use a JIT (Numba) — performance: **79 ms / frame**
3. A naive C version parallelized with OpenMP: **18.02 ms** / frame
4. A version with sophisticated optimizations (fusion + overlapped tiling): **8.97 ms** / frame (in this course, we will study how to get 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_i1 = 0; (_T_i1 <= ((R + 1) / 32)); _T_i1 = (_T_i1 + 1))
 {
   int  _ct0 = (((R + 1) < ((32 * _T_i1) + 31))? (R + 1): ((32 * _T_i1) + 31));
   int  _ct1 = ((2 > (32 * _T_i1))? 2: (32 * _T_i1));
   int  _ct4 = (((R + 1) < ((32 * _T_i1) + 31))? (R + 1): ((32 * _T_i1) + 31));
   int  _ct5 = ((2 > (32 * _T_i1))? 2: (32 * _T_i1));
   int  _ct8 = (((R + 1) < ((32 * _T_i1) + 31))? (R + 1): ((32 * _T_i1) + 31));
   int  _ct9 = ((2 > (32 * _T_i1))? 2: (32 * _T_i1));
   int  _ct12 = (((R + 1) < ((32 * _T_i1) + 31))? (R + 1): ((32 * _T_i1) + 31));
   int  _ct13 = ((2 > (32 * _T_i1))? 2: (32 * _T_i1));
   for (int  _T_i2 = -1; (_T_i2 <= ((C + 3) / 256)); _T_i2 = (_T_i2 + 1))
   {
     int  _ct2 = (((C + 3) < ((256 * _T_i2) + 261))? (C + 3): ((256 * _T_i2) + 261));
     int  _ct3 = ((0 > (256 * _T_i2))? 0: (256 * _T_i2));
     int  _ct6 = (((C + 1) < ((256 * _T_i2) + 260))? (C + 1): ((256 * _T_i2) + 260));
     int  _ct7 = ((2 > ((256 * _T_i2) + 1))? 2: ((256 * _T_i2) + 1));
     int  _ct10 = (((C + 1) < ((256 * _T_i2) + 259))? (C + 1): ((256 * _T_i2) + 259));
     int  _ct11 = ((2 > ((256 * _T_i2) + 2))? 2: ((256 * _T_i2) + 2));
     int  _ct14 = (((C + 1) < ((256 * _T_i2) + 258))? (C + 1): ((256 * _T_i2) + 258));
     int  _ct15 = ((2 > ((256 * _T_i2) + 3))? 2: ((256 * _T_i2) + 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][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _i2)] = (((((img[(((_i0 * ((R + 4) * (C + 4))) + ((-2 + _i1) * (C + 4))) + _i2)] * 0.0625f) + (img[(((_i0 * ((R + 4) * (C + 4))) + ((-1 + _i1) * (C + 4))) + _i2)] * 0.25f)) + (img[(((_i0 * ((R + 4) * (C + 4)))
+ (_i1 * (C + 4))) + _i2)] * 0.375f)) + (img[(((_i0 * ((R + 4) * (C + 4))) + ((1 + _i1) * (C + 4))) + _i2)] * 0.25f)) + (img[(((_i0 * ((R + 4) * (C + 4))) + ((2 + _i1) * (C + 4))) + _i2)] * 0.0625f));
         }
       }
     }
     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][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _i2)] = (((((blurx[_i0][((-32 * _T_i1) + _i1)][(-2 + ((-256 * _T_i2) + _i2))] * 0.0625f) + (blurx[_i0][((-32 * _T_i1) + _i1)][(-1 + ((-256 * _T_i2) + _i2))] * 0.25f))
+ (blurx[_i0][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _i2)] * 0.375f)) + (blurx[_i0][((-32 * _T_i1) + _i1)][(1 + ((-256 * _T_i2) + _i2))] * 0.25f)) + (blurx[_i0][((-32 * _T_i1) + _i1)][(2 + ((-256 * _T_i2) + _i2))] * 0.0625f));
         }
       }
     }
     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_i1) + _i1)][((-256 * _T_i2) + _i2)] = ((img[(((_i0 * ((R + 4) * (C + 4))) + (_i1 * (C + 4))) + _i2)] * (1 + weight)) + (blury[_i0][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _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_i1) + _i1)][((-256 * _T_i2) + _i2)];
           float  _ct18 = ((std::abs((img[(((_i0 * ((R + 4) * (C + 4))) + (_i1 * (C + 4))) + _i2)] - blury[_i0][((-32 * _T_i1) + _i1)][((-256 * _T_i2) + _i2)])) < threshold)? _ct16: _ct17);
           mask_flip[((((_i1-2) * (3 * C)) + ((_i2 - 2) * 3)) + (_i0))] = _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

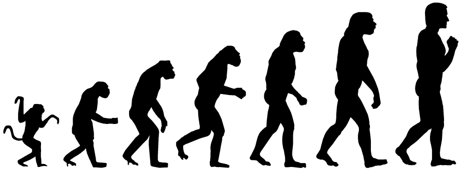
Domain-Specific

- 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



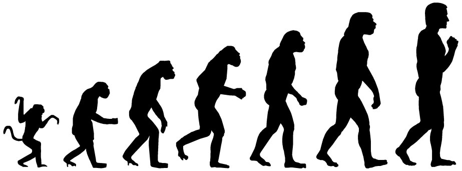
REVOLUTIONARY approach

- 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, ...

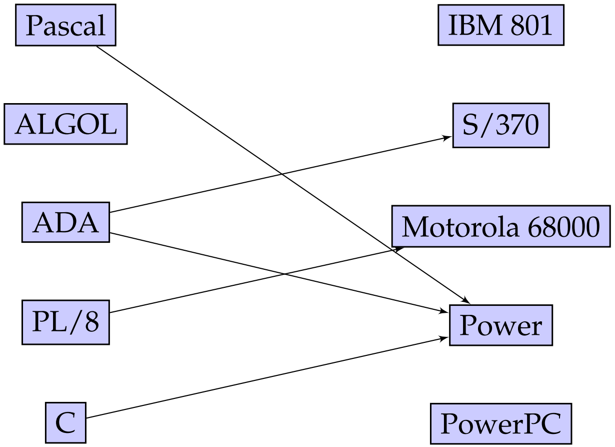
- **Both approaches share infrastructure**
- **Important to pursue both**

OUTLINE

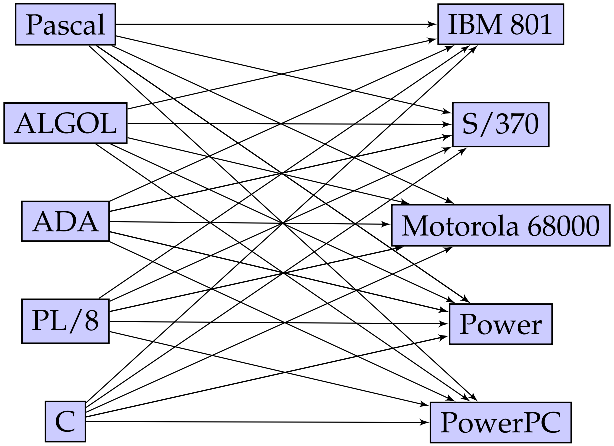
Compilers for AI

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

Compilers - The Early Days

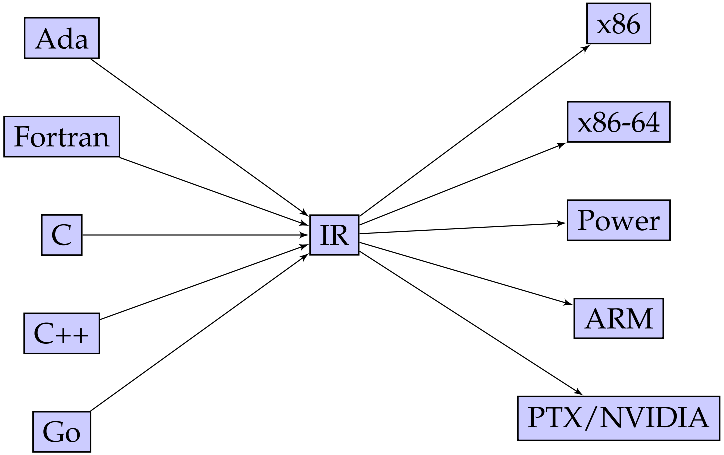


Compilers - The Early Days



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

Compilers Evolution - $M + N$



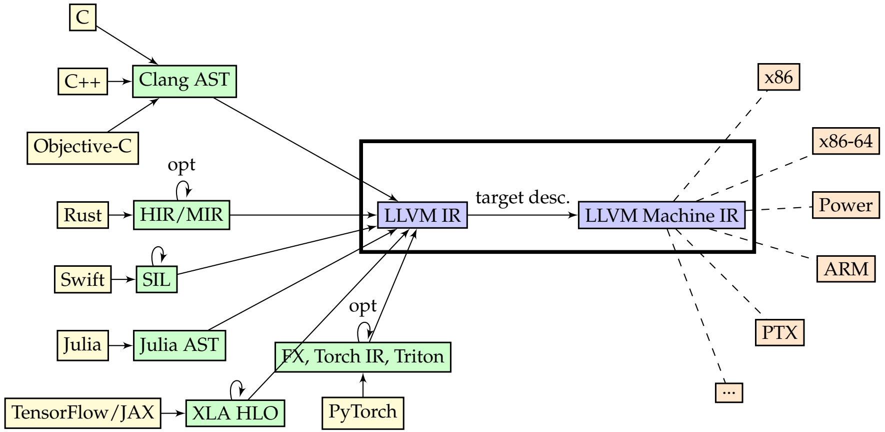
- **With an common IR, we have $M + N + 1$ compilers!**

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?

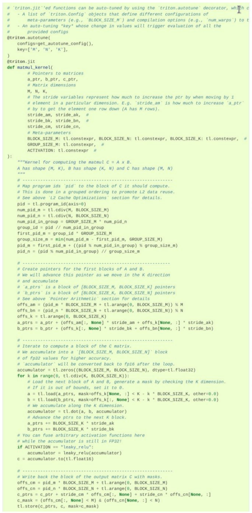
Explosion of AI chips and accelerators

- 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?

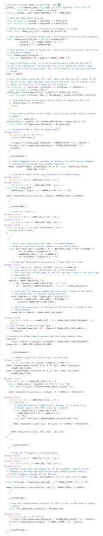
```python
# Torch function to matrix-multiply A and B.

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

PyTorch-



```python
# `triton.jit`'ed functions can be auto-tuned by using the `triton.autotune` decorator, which c
#   - 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 t
#   - 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

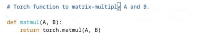


[illegible]

CUDA

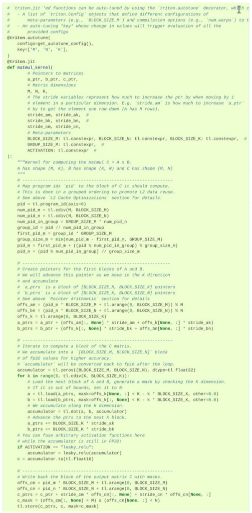
- High, mid, and low-level abstractions
- High: PyTorch, JAX, ...
- Mid: OpenAI Triton, cuTile
- Low: CUDA, C/C++, CUTLASS, ...
- All three approaches need/use compilers in different ways
- They also share/rest on the same underlying infrastructure
- Eg: Triton, MLIR, LLVM, PTX

How do you program AI hardware?



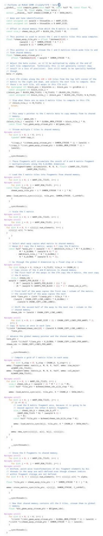
```python
# Torch function to matrix-multiply A and B.

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

PyTorch-



```python
# `triton.jit`'ed functions can be auto-tuned by using the `triton.autotune` decorator, which c
#   - 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 t
#   - 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



[illegible]

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

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

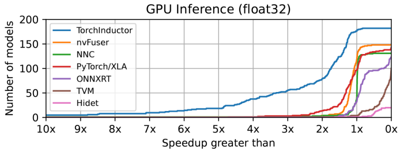
```
class SelfAttentionLayer(nn.Module):
    def __init__(self, feature_size):
        super(SelfAttentionLayer, self).__init__()
        self.feature_size = feature_size

        # 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
```

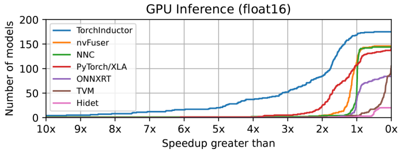
Opening multi-level doors to programming AI hardware

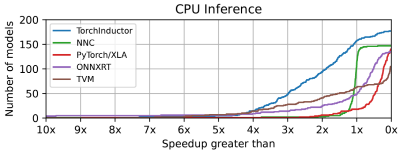
1. High-level Python-based programming frameworks (e.g. PyTorch, JAX),
2. Compiler support for (1) that could be turned off/on (e.g. torch.compile),
3. Mid/low-level programming support (e.g., CUDA, CUTLASS, CuTile, Triton)
4. Low-level MLIR dialects that expose their hardware intrinsics/virtual ISA 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.



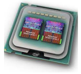


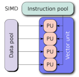


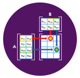
How is hardware evolving? (1/2)

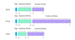
From 2000s to now

- 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)









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

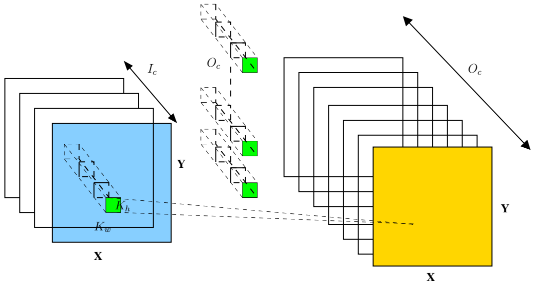
- MLIR infrastructure: open-sourced by Google in 2019

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];
```

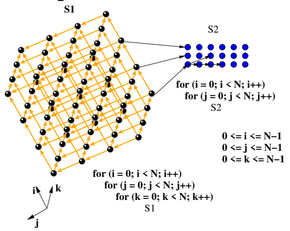


MLIR: Multi-level Intermediate Representation

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**



```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    S2

0 <= i <= N–1
0 <= j <= N–1
0 <= k <= N–1

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      S1
```

```
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)

Outline

Basic blocks and Control Flow Graph

- A **basic block** is a maximal straight-line code sequence that has a single entry point that is at its first instruction, and a single exit point that is at its last instruction.
  - Whenever the first instruction is executed, the rest of the instructions are executed exactly once, and in sequence.
  - No code within it is the target of any jump instruction
  - Only the last instruction cause control to leave the basic block
- A **control flow graph** is a directed graph where the nodes are basic blocks and the edges represent transfer of program control

Basic blocks and Control Flow Graph

- A **basic block** is a maximal straight-line code sequence that has a single entry point that is at its first instruction, and a single exit point that is at its last instruction.
  - Whenever the first instruction is executed, the rest of the instructions are executed exactly once, and in sequence.
  - No code within it is the target of any jump instruction
  - Only the last instruction cause control to leave the basic block
- A **control flow graph** is a directed graph where the nodes are basic blocks and the edges represent transfer of program control

Dominators

- A node d in a flow graph **dominates** node n, written d dom n, if every path from the initial node of the control flow graph to n goes through d
- The node x **strictly dominates** y, if x dominates y and $x \neq y$
- x is the **immediate dominator** of y, if x is the closest strict dominator of y
- A **dominator tree** shows all the immediate dominator relationships
- How do you find the dominators?
    - What can you say about a node's predecessors' dominators and its dominators?

DOMINATORS

- A node d in a flow graph **dominates** node n, written d dom n, if every path from the initial node of the control flow graph to n goes through d
- The node x **strictly dominates** y, if x dominates y and $x \neq y$
- x is the **immediate dominator** of y, if x is the closest strict dominator of y
- A **dominator tree** shows all the immediate dominator relationships
- How do you find the dominators?
  - What can you say about a node's predecessors' dominators and its dominators?

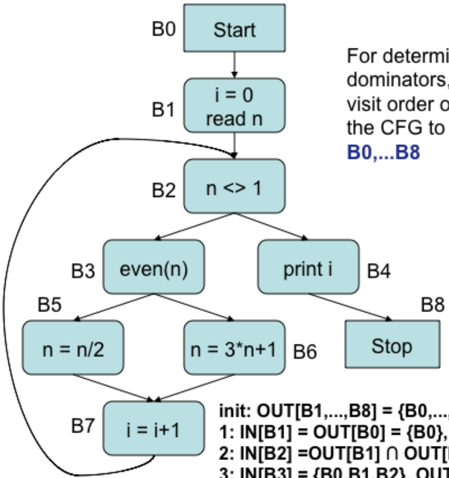
Dominators

- A node d in a flow graph **dominates** node n, written d dom n, if every path from the initial node of the control flow graph to n goes through d
- The node x **strictly dominates** y, if x dominates y and $x \neq y$
- x is the **immediate dominator** of y, if x is the closest strict dominator of y
- A **dominator tree** shows all the immediate dominator relationships
- How do you find the dominators?
    - What can you say about a node's predecessors' dominators and its dominators?

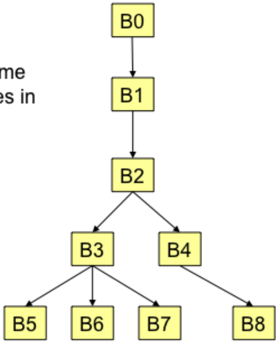
Algorithm to Find Dominators

```
**Input** : CFG $(V, E)$
**Output:** Dom(n) $\forall n \in V$
$Dom(s) \leftarrow \{s\}$
for each $n \in V - \{s\}$ do
    $Dom(n) \leftarrow V$
while changes to Dom(n) occur do
    for each $n \in V - \{s\}$ do
        $IN(n) \leftarrow \cap_{p \in pred(n)} Dom(p)$
    for each $n \in V - \{s\}$ do
        $Dom(n) \leftarrow \{n\} \cup IN(n)$
```

Dominator Example



For determining dominators, assume visit order of nodes in the CFG to be **B0,...B8**

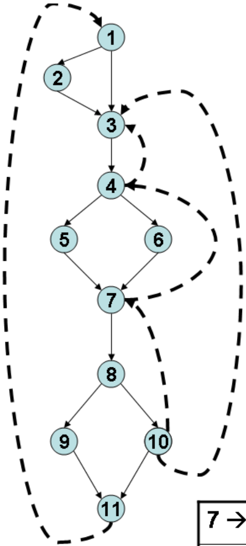


init: OUT[B1,...,B8] = {B0,...,B8}, OUT[B0] = {B0}
1: IN[B1] = OUT[B0] = {B0}, OUT[B1] = {B0,B1}
2: IN[B2] =OUT[B1] ∩ OUT[B7] = {B0,B1} , OUT[B2] = {B0,B1,B2}
3: IN[B3] = {B0,B1,B2}, OUT[B3] = {B0,B1,B2,B3}
IN[B4] = {B0,B1,B2}, OUT[B4] = {B0,B1,B2,B4}
4: IN[B5] = {B0,B1,B2,B3} = IN[B6], OUT[B5] = {B0,B1,B2,B3,B5}
OUT[B6] = {B0,B1,B2,B3,B6}, OUT[B8] = {B0,B1,B2,B4,B8}
5: IN[B7] = OUT[B5] ∩ OUT[B6] = {B0,B1,B2,B3}
OUT[B7] = {B0,B1,B2,B3,B7}

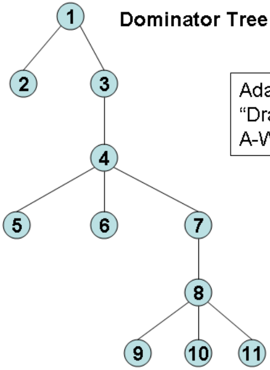
DFS on CFG, Back Edges, Natural Loops

- A DFS will yield tree edges, forward edges, cross edges, and **retreating** edges — what are these?
- A **back edge** in a CFG is an edge whose head dominates the tail (definition is not the same as the one used for depth first search in graphs)
- A **natural loop** of a back edge is the set of nodes comprising the head node of the back edges and the nodes that can reach the tail of the back edge without going through the head node
    - It is intuitively the region of the program that may be executed iteratively/repeatedly with the back edge being used for looping and with the head of the back edge as the *only* entry point to this region
    - It is intuitively the loop body of a for loop or a similar iterative construct (while, do/while)

Dominators, Back Edges, and Natural Loops



Flow Graph

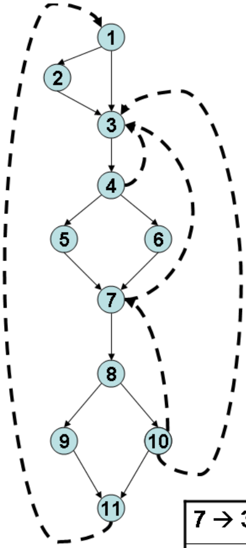


Adapted from the "Dragon Book", A-W, 1986

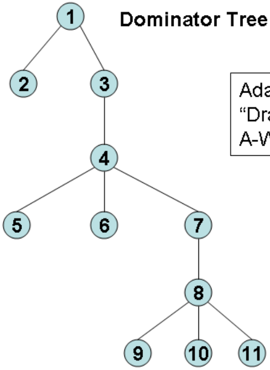
Back edges and their natural loops

7 → 4	10→7	4→3	10→3	11→1
{4,5,6,7,8, 10}	{7,8,10}	{3,4,5,6,7, 8,10}	{3,4,5,6,7, 8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

Dominators, Back Edges, and Natural Loops



Flow Graph



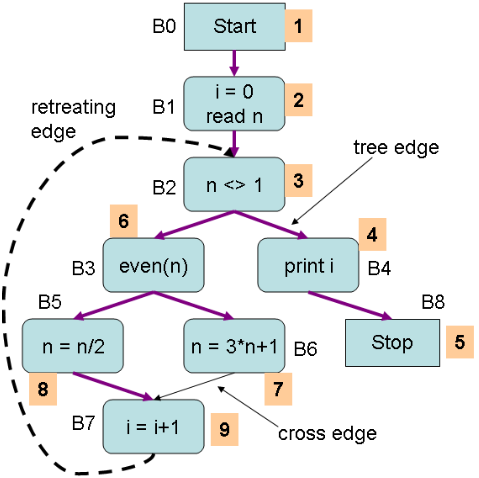
Dominator Tree

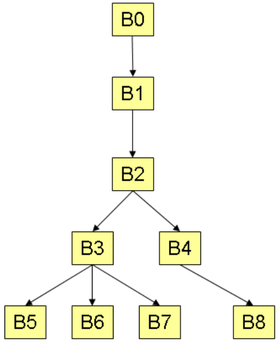
Adapted from the "Dragon Book", A-W, 1986

Back edges and their natural loops

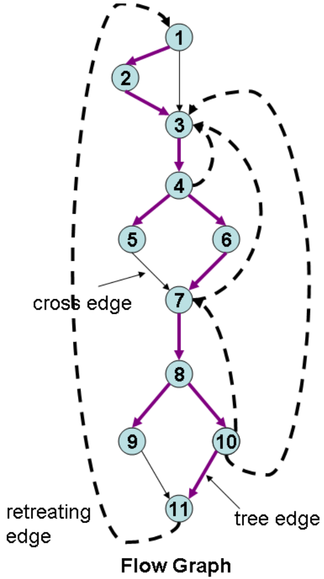
7 → 3	10→7	4→3	10→3	11→1
{3,4,5,6,7,8, 10}	{7,8,10}	{3,4}	{3,4,5,6,7, 8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

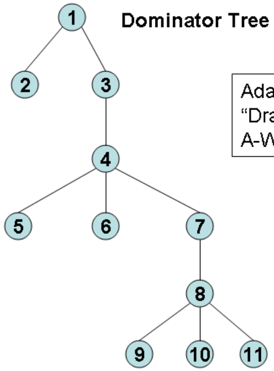
Depth-First Numbering Example 1





Depth-First Numbering Example 2





Adapted from the "Dragon Book", A-W, 1986

Nodes of the CFG show the DF-numbering

Control Flow Graph Reducibility

- A control flow graph is reducible if all its retreating edges are back edges
- A control flow graph is reducible if if it can be reduced to a single node by repeatedly applying *T*1 and *T*2 transformations
  - T1: Eliminate a self loop
  - T2: Merge a single entry node into its parent
- Are your flow graphs reducible?
- What about structured programming?

CONTROL FLOW GRAPH REDUCIBILITY

- A control flow graph is reducible if all its retreating edges are back edges
- A control flow graph is reducible if if it can be reduced to a single node by repeatedly applying $T1$ and $T2$ transformations
    - T1: Eliminate a self loop
    - T2: Merge a single entry node into its parent
- Are your flow graphs reducible?
- What about structured programming?

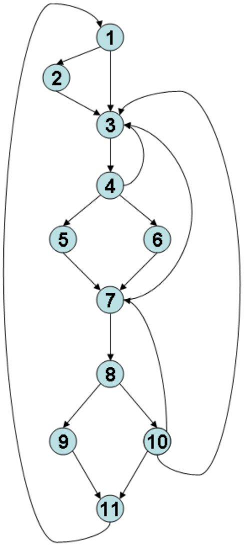
Control Flow Graph Reducibility

- A control flow graph is reducible if all its retreating edges are back edges
- A control flow graph is reducible if if it can be reduced to a single node by repeatedly applying $T1$ and $T2$ transformations
    - T1: Eliminate a self loop
    - T2: Merge a single entry node into its parent
- Are your flow graphs reducible?
- What about structured programming?

References

- *Control flow Analsis, Frances Allen, 1970.*
  `http://dl.acm.org/citation.cfm?id=808479`

Reducibility - Example 1



Flow Graph

7 → 3, 10 → 7, 4 → 3, 10 → 3, and 11 → 1 are all back edges.

There are no other retreating edges in any depth-first search tree of this graph.

The rest of the edges form a DAG, in which each node is reachable from node 1.

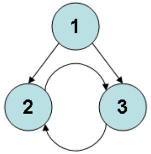
Reducible graph.

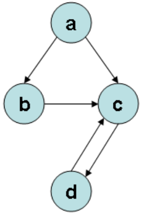
Reducibility - Example 2

Irreducible graph, no back edge.

Either 2 → 3 or 3 → 2 is a retreating edge in a depth-first search tree.

The graph is cyclic, not a DAG.





d → c is a back edge.

Other edges form a DAG in which each node is reachable from the node a.

Reducible graph.

Outline

Data-flow analysis

- These are techniques that derive information about the flow of data along program execution paths
- An *execution path* (or *path*) from point p_1 to point p_n is a sequence of points $p_1, p_2, ..., p_n$ such that for each $i = 1, 2, ..., n-1$, either
  1. p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or
  2. p_i is the end of some block and p_{i+1} is the beginning of a successor block
- In general, there is an infinite number of paths through a program and there is no bound on the length of a path
- Program analyses summarize all possible program states that can occur at a point in the program with a finite set of facts
- No analysis is necessarily a perfect representation of the state

Uses of Data-flow Analysis

- Program debugging
  - Which are the definitions (of variables) that *may* reach a program point? These are the *reaching definitions*
- Program optimizations
  - Constant folding
  - Copy propagation
  - Common sub-expression elimination etc.

Data-Flow Analysis Schema

- A *data-flow value* for a program point represents an abstraction of the set of all possible program states that can be observed for that point
- The set of all possible data-flow values is the *domain* for the application under consideration
    - Example: for the *reaching definitions* problem, the domain of data-flow values is the set of all subsets of of definitions in the program
    - A particular data-flow value is a set of definitions
- *IN*[*s*] and *OUT*[*s*]: data-flow values *before* and *after* each statement *s*
- The *data-flow problem* is to find a solution to a set of constraints on *IN*[*s*] and *OUT*[*s*], for all statements *s*

The Reaching Definitions Problem

- We *kill* a definition of a variable *a*, if between two points along the path, there is an assignment to *a*
- A definition *d* reaches a point *p*, if there is a path from the point immediately following *d* to *p*, such that *d* is not *killed* along that path
- Unambiguous and ambiguous definitions of a variable

  ```
      a := b+c
  ```
  (unambiguous definition of ’a’)
  ```
      ...
      *p := d
  ```
  (ambiguous definition of ’a’, if ’p’ may point to variables other than ’a’ as well; hence does not kill the above definition of ’a’)
  ```
      ...
      a := k-m
  ```
  (unambiguous definition of ’a’; kills the above definition of ’a’)

The Reaching Definitions Problem(2)

- We compute supersets of definitions as *safe* values
- It is safe to assume that a definition reaches a point, even if it does not.
- In the following example, we assume that both `a=2` and `a=4` reach the point after the complete if-then-else statement, even though the statement `a=4` is not reached by control flow

  ```
  if (a==b) a=2; else if (a==b) a=4;
  ```

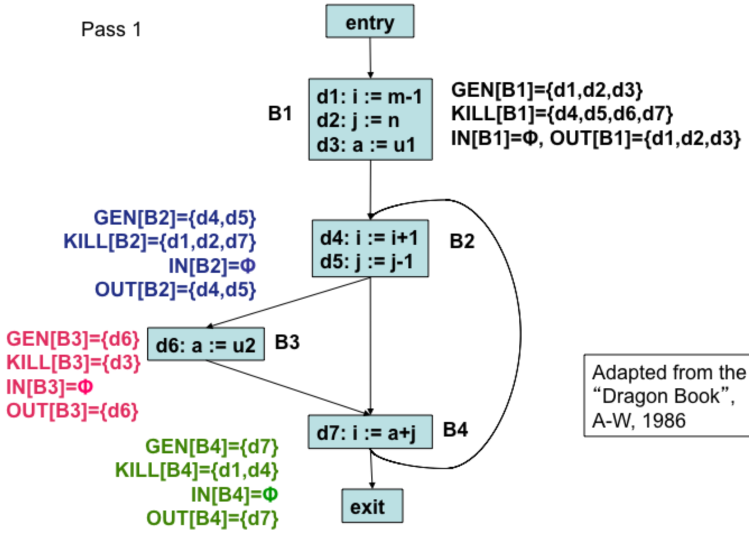
The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$\begin{aligned} IN[B] &= \bigcup_{P\ is\ a\ predecessor\ of\ B} OUT[P] \\ OUT[B] &= GEN[B] \bigcup (IN[B] - KILL[B]) \\ IN[B] &= \phi, \textit{for all } B \ (\textit{initialization only}) \end{aligned}$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
- $GEN[B]$ = set of all definitions inside B that are "visible" immediately after the block - *downwards exposed* definitions
- $KILL[B]$ = union of the definitions in all the basic blocks of the flow graph, that are killed by individual statements in B

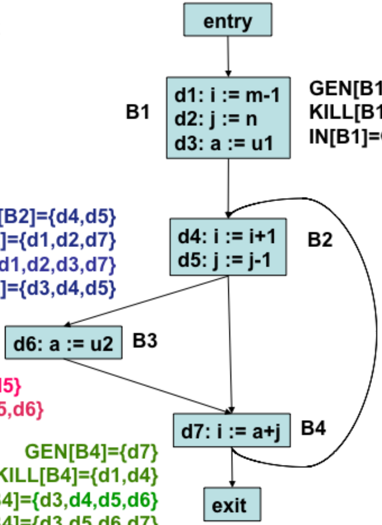
Reaching Definitions Analysis: An Example - Pass 1



Adapted from the "Dragon Book", A-W, 1986

Reaching Definitions Analysis: An Example - Pass 2

Pass 2



GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]={d1,d2,d3,d7}
OUT[B2]={d3,d4,d5}

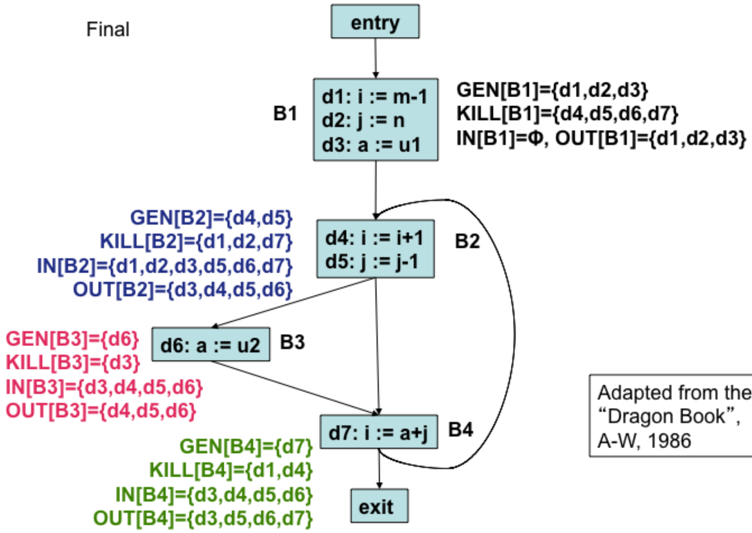
GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]={d3,d4,d5}
OUT[B3]={d4,d5,d6}

GEN[B4]={d7}
KILL[B4]={d1,d4}
IN[B4]={d3,d4,d5,d6}
OUT[B4]={d3,d5,d6,d7}

Adapted from the "Dragon Book", A-W, 1986

Reaching Definitions Analysis: An Example - Final

Final



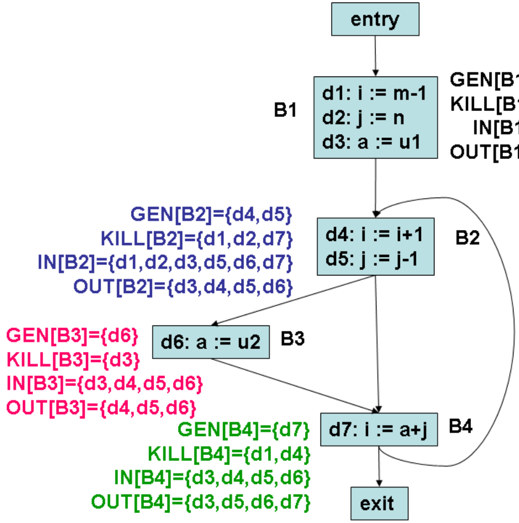
Adapted from the "Dragon Book", A-W, 1986

An Iterative Algorithm for Computing Reaching Definitions

```
for each block $B$ do { $IN[B] = \phi$; $OUT[B] = GEN[B]$; }
$change = true$;
while $change$ do { $change = false$;
  for each block $B$ do {
      $IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P]$;
      $oldout = OUT[B]$;
      $OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$;
    if ($OUT[B] \neq oldout$) $change = true$;
  }
}
```

- GEN, $KILL$, IN, and OUT are all represented as bit vectors with one bit for each definition in the flow graph

Reaching Definitions: Bit Vector Representation



Final dataflow value sets shown in bit vector format

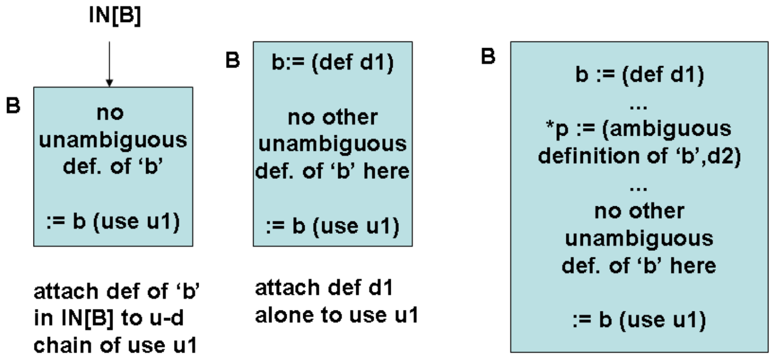
GEN[B1]=	1	1	1	0	0	0	0
KILL[B1]=	0	0	0	1	1	1	1
IN[B1]=	0	0	0	0	0	0	0
OUT[B1]=	1	1	1	0	0	0	0
	d1	d2	d3	d4	d5	d6	d7

Adapted from the "Dragon Book", A-W, 1986

Use-Definition Chains (u-d chains)

- Reaching definitions may be stored as u-d chains for convenience
- A u-d chain is a list of a use of a variable and all the definitions that reach that use
- u-d chains may be constructed once reaching definitions are computed
- **case 1**: If use *u*1 of a variable *b* in block B is preceded by no unambiguous definition of *b*, then attach all definitions of *b* in *IN*[*B*] to the u-d chain of that use *u*1 of *b*
- **case 2**: If any unambiguous definition of *b* preceeds a use of *b*, then *only that definition* is on the u-d chain of that use of *b*
- **case 3**: If any ambiguous definitions of *b* precede a use of *b*, then each such definition for which no unambiguous definition of *b* lies between it and the use of *b*, are on the u-d chain for this use of *b*

Use-Definition Chain Construction


IN[B]
B
no unambiguous def. of ‘b’
:= b (use u1)
B
b:= (def d1)
no other unambiguous def. of ‘b’ here
:= b (use u1)
B
b := (def d1)
...
*p := (ambiguous definition of ‘b’,d2)
...
no other unambiguous def. of ‘b’ here
:= b (use u1)

attach def of ‘b’ in IN[B] to u-d chain of use u1

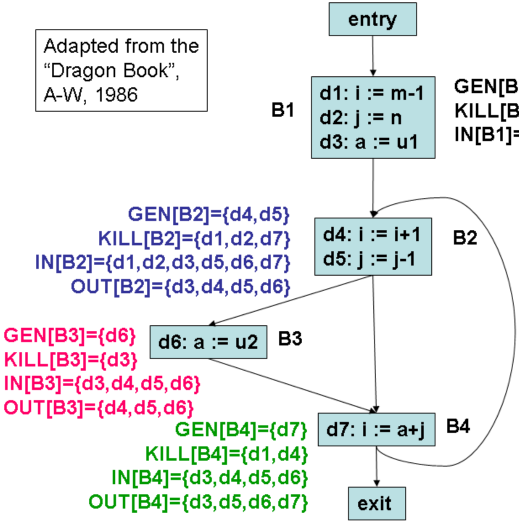
attach def d1 alone to use u1

attach both d1 and d2 to use u1

Three cases while constructing u-d chains from the reaching definitions

Use-Definition Chain Example

Adapted from the "Dragon Book", A-W, 1986



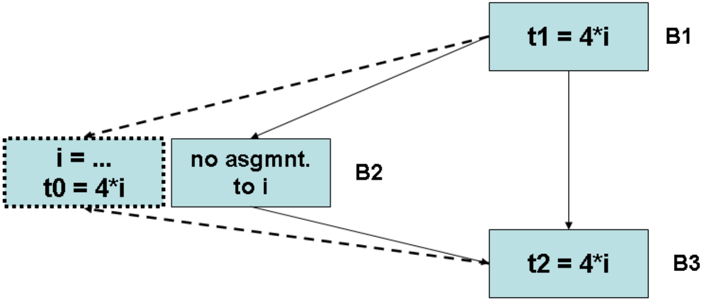
use	u-d chain
(i,d4)	(d1,d7)
(j,d5)	(d2,d5)
(a,d7)	(d3,d6)
(j,d7)	(d5)

Available Expression Computation

- Sets of expressions constitute the domain of data-flow values
- Forward flow problem
- Confluence operator is $\cap$
- An expression $x + y$ is *available* at a point p, if every path (not necessarily cycle-free) from the initial node to p evaluates $x + y$, and after the last such evaluation, prior to reaching p, there are no subsequent assignments to x or y
- A block *kills* $x + y$, if it assigns (or may assign) to x or y and does not subsequently recompute $x + y$.
- A block *generates* $x + y$, if it definitely evaluates $x + y$, and does not subsequently redefine x or y

Available Expression Computation(2)

- Useful for global common sub-expression elimination
- $4 * i$ is a CSE in $B3$, if it is available at the entry point of $B3$ *i.e.,* if i is not assigned a new value in $B2$ or $4 * i$ is recomputed after i is assigned a new value in $B2$ (as shown in the dotted box)



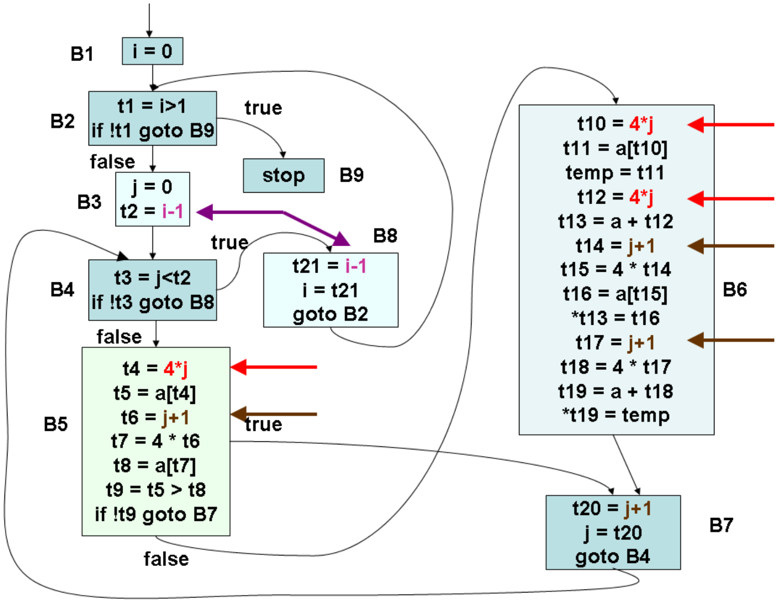
Available Expression Computation (3)

- The data-flow equations

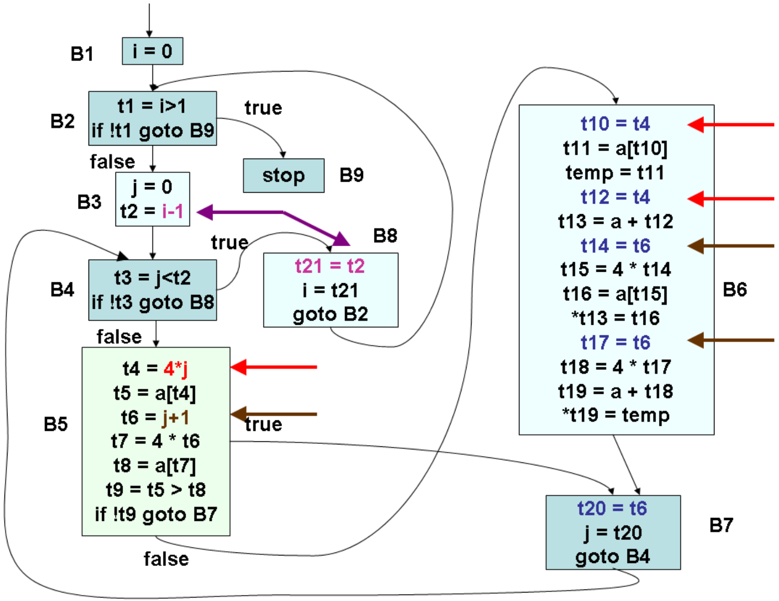
$$\begin{aligned}
IN[B] &= \bigcap_{P\ is\ a\ predecessor\ of\ B} OUT[P],\ B\ not\ initial \\
OUT[B] &= e\_gen[B] \bigcup (IN[B] - e\_kill[B]) \\
IN[B1] &= \phi \\
IN[B] &= U,\ for\ all\ B \neq B1\ (initialization\ only)
\end{aligned}$$

- $B1$ is the intial or entry block and is special because nothing is available when the program begins execution
- $IN[B1]$ is always ϕ
- U is the universal set of all expressions
- Initializing $IN[B]$ to ϕ for all $B \neq B1$, is restrictive

Available Expression Computation - An Example



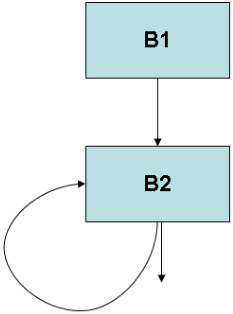
Available Expression Computation - An Example (2)



An Iterative Algorithm for Computing Available Expressions

```
for each block $B \neq B1$ do {$OUT[B] = U - e\_kill[B];$ }
/* You could also do $IN[B] = U$;*/
/* In such a case, you must also interchange the order of */
/* $IN[B]$ and $OUT[B]$ equations below */
$change = true$;
while $change$ do { $change = false$;
  for each block $B \neq B1$ do {
        $IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P];$
        $oldout = OUT[B];$
        $OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B]);$
    if ($OUT[B] \neq oldout$) $change = true$;
  }
}
```

Initializing IN[B] to ϕ for all B can be restrictive



Let e_gen[B2] be G and e_kill[B2] be K

IN[B2] = OUT[B1] ∩ OUT[B2]

OUT[B2] = G U (IN[B2] – K)

$IN^0[B2]=\Phi$, $OUT^1[B2]=G$

$IN^1[B2]=OUT[B1] \cap G$

$OUT^2[B2]=G \cup ((OUT[B1] \cap G) - K)$

$= G \cup G = G$

Note that (OUT[B1] ∩ G) is always smaller than G

$IN^0[B2]= \boldsymbol{U}$, $OUT^1[B2]= \boldsymbol{U} - K$

$IN^1[B2]=OUT[B1] \cap (\boldsymbol{U} - K)$

$= OUT[B1] - K$

$OUT^2[B2]=G \cup ((OUT[B1] - K) - K)$

$= G \cup (OUT[B1] - K)$

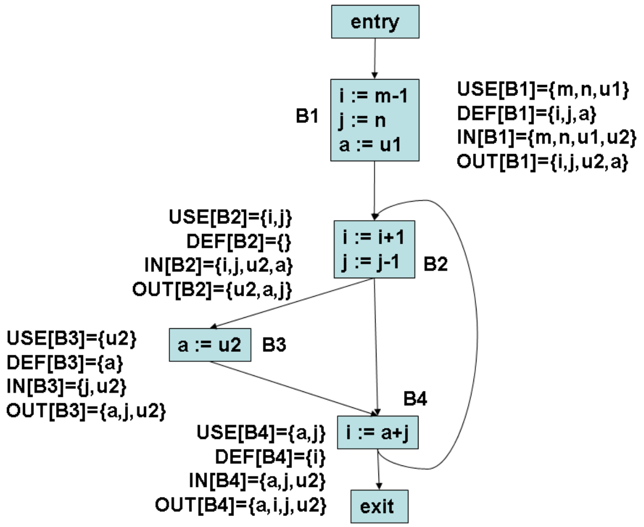
This set OUT[B2] is larger and more intuitive, but still correct

Live Variable Analysis

- The variable x is *live* at the point p, if the value of x at p could be used along some path in the flow graph, starting at p; otherwise, x is *dead* at p
- Sets of variables constitute the domain of data-flow values
- Backward flow problem, with confluence operator $\bigcup$
- $IN[B]$ is the set of variables live at the beginning of B
- $OUT[B]$ is the set of variables live just after B
- $DEF[B]$ is the set of variables definitely assigned values in B, prior to any use of that variable in B
- $USE[B]$ is the set of variables whose values may be used in B prior to any definition of the variable

$$\begin{aligned} OUT[B] &= \bigcup_{S \textit{ is a successor of } B} IN[S] \\ IN[B] &= USE[B] \bigcup (OUT[B] - DEF[B]) \\ IN[B] &= \phi, \textit{for all } B \textit{ (initialization only)} \end{aligned}$$

Live Variable Analysis: An Example



Definition-Use Chains (d-u chains)

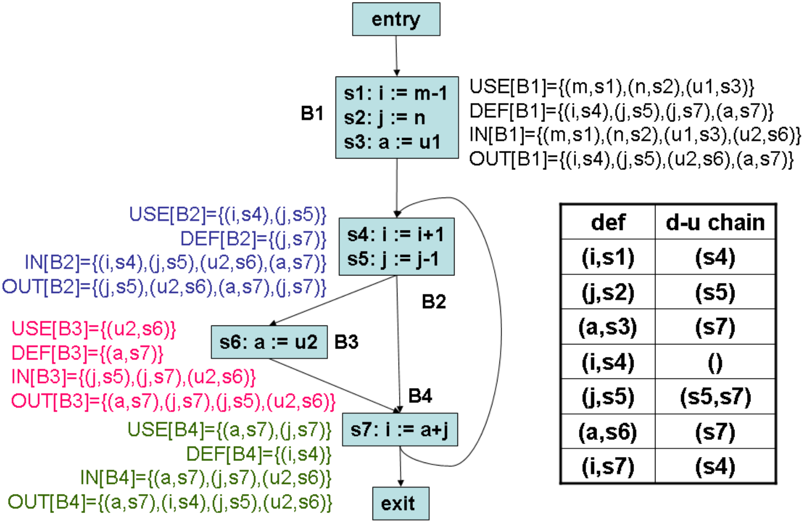
- For each definition, we wish to attach the statement numbers of the uses of that definition
- Such information is very useful in implementing register allocation, loop invariant code motion, etc.
- This problem can be transformed to the data-flow analysis problem of computing for a point p, the set of uses of a variable (say x), such that there is a path from p to the use of x, that does not redefine x.
- This information is represented as sets of (x, s) pairs, where x is the variable used in statement s
- In live variable analysis, we need information on whether a variable is used later, but in (x, s) computation, we also need the statment numbers of the uses
- The data-flow equations are similar to that of LV analysis
- Once $IN[B]$ and $OUT[B]$ are computed, d-u chains can be computed using a method similar to that of u-d chains

Data-flow Analysis for (x,s) pairs

- Sets of pairs (x,s) constitute the domain of data-flow values
- Backward flow problem, with confluence operator $\bigcup$
- $USE[B]$ is the set of pairs (x, s), such that s is a statement in B which uses variable x and such that no prior definition of x occurs in B
- $DEF[B]$ is the set of pairs (x, s), such that s is a statement which uses x, s is *not in* B, and B contains a definition of x
- $IN[B]$ ($OUT[B]$, resp.) is the set of pairs (x, s), such that statement s uses variable x and the value of x at $IN[B]$ ($OUT[B]$, resp.) has not been modified along the path from $IN[B]$ ($OUT[B]$, resp.) to s

$$\begin{aligned} OUT[B] &= \bigcup_{S \text{ is a successor of } B} IN[S] \\ IN[B] &= USE[B] \bigcup (OUT[B] - DEF[B]) \\ IN[B] &= \phi, \textit{ for all } B \textit{ (initialization only)} \end{aligned}$$

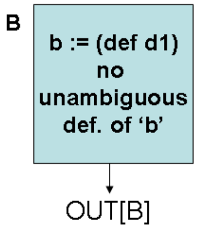
Definition-Use Chain Example



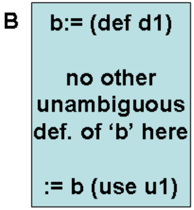
entry
B1
s1: i := m-1
s2: j := n
s3: a := u1
USE[B1]={(m,s1),(n,s2),(u1,s3)}
DEF[B1]={(i,s4),(j,s5),(j,s7),(a,s7)}
IN[B1]={(m,s1),(n,s2),(u1,s3),(u2,s6)}
OUT[B1]={(i,s4),(j,s5),(u2,s6),(a,s7)}
USE[B2]={(i,s4),(j,s5)}
DEF[B2]={(j,s7)}
IN[B2]={(i,s4),(j,s5),(u2,s6),(a,s7)}
OUT[B2]={(j,s5),(u2,s6),(a,s7),(j,s7)}
s4: i := i+1
s5: j := j-1
B2
USE[B3]={(u2,s6)}
DEF[B3]={(a,s7)}
IN[B3]={(j,s5),(j,s7),(u2,s6)}
OUT[B3]={(a,s7),(j,s7),(j,s5),(u2,s6)}
s6: a := u2
B3
B4
USE[B4]={(a,s7),(j,s7)}
DEF[B4]={(i,s4)}
IN[B4]={(a,s7),(j,s7),(u2,s6)}
OUT[B4]={(a,s7),(i,s4),(j,s5),(u2,s6)}
s7: i := a+j
exit

def	d-u chain
(i,s1)	(s4)
(j,s2)	(s5)
(a,s3)	(s7)
(i,s4)	()
(j,s5)	(s5,s7)
(a,s6)	(s7)
(i,s7)	(s4)

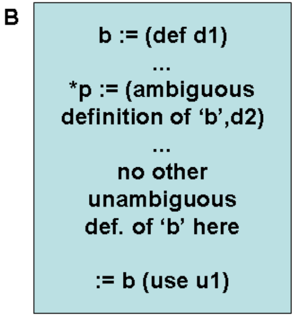
Definition-Use Chain Construction



attach to du-chain of d1, stmts s_i of all use pairs (b,s_i) in OUT[B]



attach use u1 to du-chain of def d1

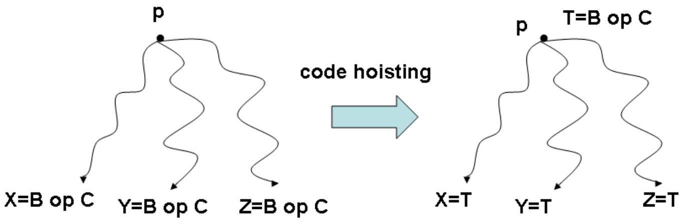


attach use u1 to du-chains of both def d1 and def d2

Three cases while constructing d-u chains from the (x,s) pairs

Very Busy Expressions *or* Anticipated Expressions

- An expression *B op C* is very busy or anticipated at a point *p*, if along every path from *p*, we come to a computation of *B op C* before any computation of *B* or *C*
- Useful in code hoisting and partial redundancy elimination
- Code hoisting does not reduce time, but reduces space
- We must make sure that no use of *B op C* (from X,Y, or Z below) has any definition of *B* or *C* reaching it without passing through *p*

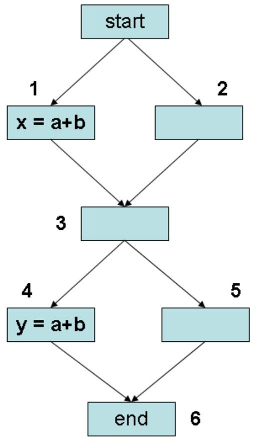


Very Busy Expressions *or* Anticipated Expressions (2)

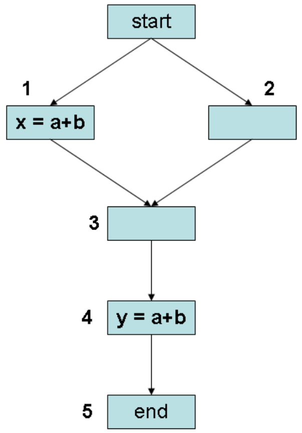
- Sets of expressions constitute the domain of data-flow values
- Backward flow analysis with $\bigcap$ as confluence operator
- $V\_USE[n]$ is the set of expressions $B\ op\ C$ computed in n with no prior definition of B or C in n
- $V\_DEF[n]$ is the set of expressions $B\ op\ C$ in U (the universal set of expressions) for which either B or C is defined in n, prior to any computation of $B\ op\ C$

$$\begin{aligned}
OUT[n] &= \bigcap_{S\ is\ a\ successor\ of\ n} IN[S] \\
IN[n] &= V\_USE[n] \bigcup (OUT[n] - V\_DEF[n]) \\
IN[n] &= U, \mathit{for\ all\ n\ (initialization\ only)}
\end{aligned}$$

Anticipated Expressions - An Example



(a)



(b)

a+b is anticipated at: entry to 1 and 4
a+b is not anticipated at: all other points

a+b is anticipated at all points,
except at exit of 4 and entry of 5

Data-Flow Problems: A Summary - 1

The Reaching Definitions Problem

- Domain of data-flow values: sets of definitions
- Direction: Forwards
- Confluence operator: $\cup$
- Initialization: $IN[B] = \phi$
- Equations:

$$\begin{aligned} IN[B] &= \bigcup_{P\ is\ a\ predecessor\ of\ B} OUT[P] \\ OUT[B] &= GEN[B] \bigcup (IN[B] - KILL[B]) \end{aligned}$$

Data-Flow Problems: A Summary - 2

The Available Expressions Problem

- Domain of data-flow values: sets of expressions
- Direction: Forwards
- Confluence operator: $\cap$
- Initialization: $IN[B] = U$
- Equations:

$$\begin{aligned} IN[B] &= \bigcap_{P \ is \ a \ predecessor \ of \ B} OUT[P] \\ OUT[B] &= e\_gen[B] \bigcup (IN[B] - e\_kill[B]) \\ IN[B1] &= \phi \end{aligned}$$

Data-Flow Problems: A Summary - 3

The Live Variable Analysis Problem

- Domain of data-flow values: sets of variables
- Direction: backwards
- Confluence operator: $\cup$
- Initialization: $IN[B] = \phi$
- Equations:

$$
\begin{aligned}
OUT[B] &= \bigcup_{S \text{ is a successor of } B} IN[S] \\
IN[B] &= USE[B] \bigcup (OUT[B] - DEF[B])
\end{aligned}
$$

Data-Flow Problems: A Summary - 4

The Anticipated Expressions (Very Busy Expressions) Problem

- Domain of data-flow values: sets of expressions
- Direction: backwards
- Confluence operator: $\cap$
- Initialization: $IN[B] = U$
- Equations:

$$\begin{aligned} OUT[B] &= \bigcap_{S \text{ is a successor of } B} IN[S] \\ IN[B] &= V\_USE[B] \bigcup (OUT[B] - V\_DEF[B]) \end{aligned}$$

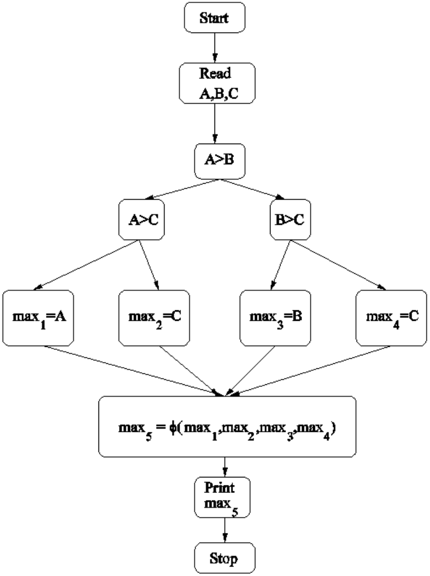
Outline

The SSA Form: Introduction

- A new intermediate representation
- Incorporates *def-use* information
- Every variable has exactly one definition in the program text
    - This does not mean that there are no loops
    - This is a *static* single assignment form, and not a *dynamic* single assignment form
- Some compiler optimizations perform better on SSA forms
    - Conditional constant propagation and global value numbering are faster and more effective on SSA forms
- A *sparse* intermediate representation
    - If a variable has N uses and M definitions, then *def-use chains* need space and time proportional to $N.M$
    - But, the corresponding instructions of uses and definitions are only $N + M$ in number
    - SSA form, for most realistic programs, is linear in the size of the original program

A Program in non-SSA Form and its SSA Form

```
read A,B,C
if (A>B)
   if (A>C) max = A
   else max = C
else if (B>C) max = B
      else max = C
printf (max)
```



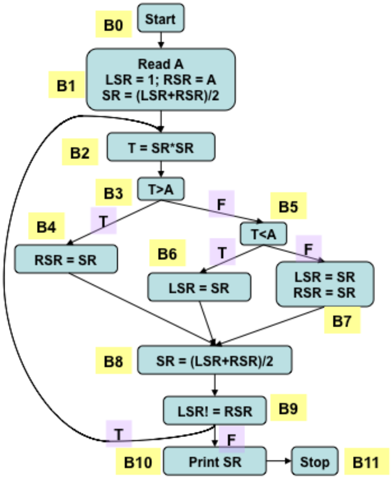
SSA Form: A Definition

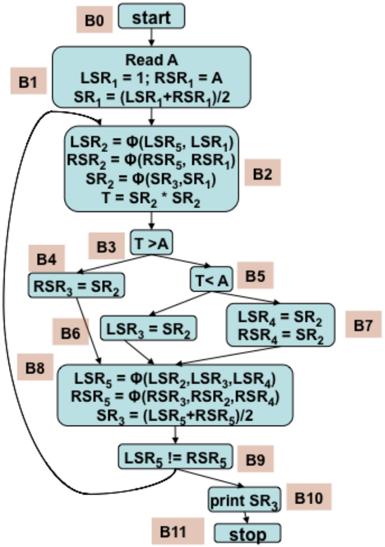
- A program is in SSA form, if each use of a variable is reached by exactly one definition
- Flow of control remains the same as in the non-SSA form
- A special merge operator, ϕ, is used for selection of values in join nodes
- Not every join node needs a ϕ operator for every variable
- No need for a ϕ operator, if the same definition of the variable reaches the join node along all incoming edges
- Often, an SSA form is augmented with *u-d* and *d-u* chains to facilitate design of faster algorithms
- Translation from SSA to machine code introduces copy operations, which may introduce some inefficiency

Program 2 in non-SSA Text Form

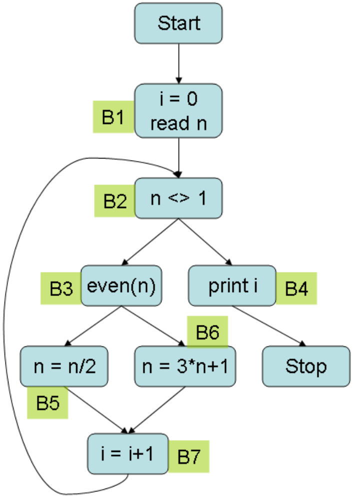
```
{ Read A; LSR = 1; RSR = A;
  SR = (LSR+RSR)/2;
  Repeat {
        T = SR*SR;
        if (T>A) RSR = SR;
        else if (T<A) LSR = SR;
                else { LSR = SR; RSR = SR}
        SR = (LSR+RSR)/2;
  Until (LSR ≠ RSR);
  Print SR;
}
```

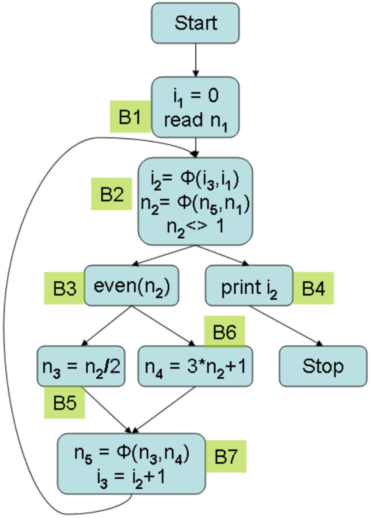
Program 2 in non-SSA and SSA Form





Program 3 in non-SSA and SSA Form





Conditions on the SSA form

After translation, the SSA form should satisfy the following conditions for every variable v in the original program.

1. If two non-null paths from nodes X and Y each having a definition of v converge at a node p, then p contains a trivial ϕ-function of the form $v = \phi(v, v, ..., v)$, with the number of arguments equal to the in-degree of p.
2. Each appearance of v in the original program or a ϕ-function in the new program has been replaced by a new variable v_i, leaving the new program in SSA form.
3. Any use of a variable v along any control path in the original program and the corresponding use of v_i in the new program yield the same value for both v and v_i.

Conditions on SSA Forms

- Condition 1 in the previous slide is recursive.
  - It implies that ϕ-assignments introduced by the translation procedure will also qualify as assignments to *v*
  - This in turn may lead to introduction of more ϕ-assignments at other nodes
- It would be wasteful to place ϕ-functions in all join nodes
- It is possible to locate the nodes where ϕ-functions are *essential*
- This is captured by the *dominance frontier*

The Join Sets and ϕ Nodes

Given $\mathcal{S}$: set of flow graph nodes, the set *JOIN*($\mathcal{S}$) is

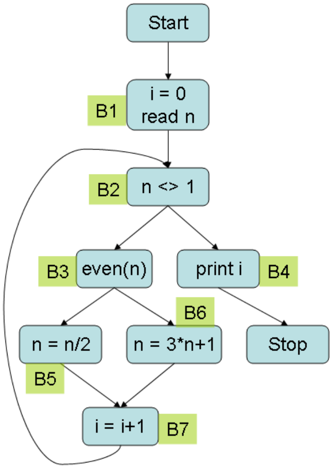
- the set of all nodes *n*, such that there are at least two non-null paths in the flow graph that start at two distinct nodes in $\mathcal{S}$ and converge at *n*
    - The paths considered should not have any other common nodes apart from *n*
- The *iterated join set*, $JOIN^{+}(\mathcal{S})$ is

$$\begin{aligned} JOIN^{(1)}(\mathcal{S}) &= JOIN(\mathcal{S}) \\ JOIN^{(i+1)}(\mathcal{S}) &= JOIN(\mathcal{S} \cup JOIN^{(i)}(\mathcal{S})) \end{aligned}$$

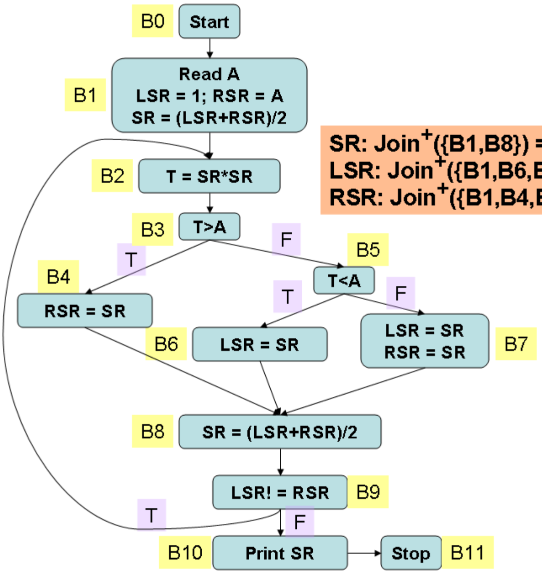
- If $\mathcal{S}$ is the set of assignment nodes for a variable *v*, then $JOIN^{+}(\mathcal{S})$ is precisely the set of flow graph nodes, where ϕ-functions are needed (for *v*)
- $JOIN^{+}(\mathcal{S})$ is termed the *dominance frontier*, $DF(\mathcal{S})$, and can be computed efficiently

JOIN Example -1

- variable i: $JOIN^{+}(\{B1, B7\}) = \{B2\}$
- variable n: $JOIN^{+}(\{B1, B5, B6\}) = \{B2, B7\}$



JOIN Example - 2



SR: $\text{Join}^{+}(\{B1,B8\}) = \{B2\}$
LSR: $\text{Join}^{+}(\{B1,B6,B7\}) = \{B2,B8\}$
RSR: $\text{Join}^{+}(\{B1,B4,B7\}) = \{B2,B8\}$

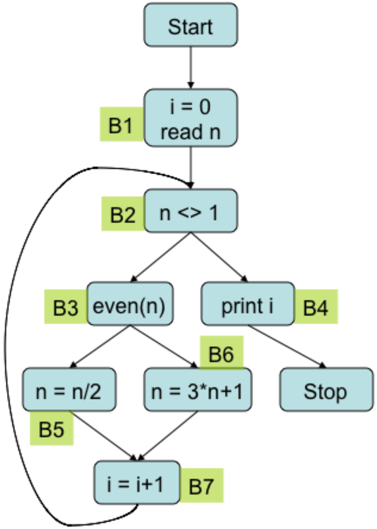
Dominators and Dominance Frontier

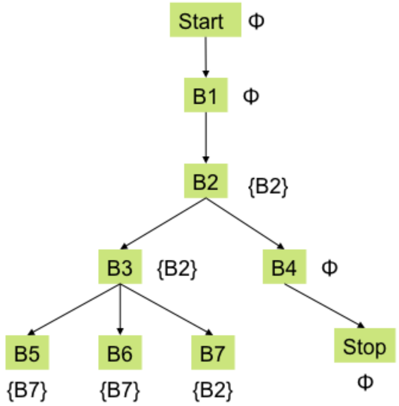
- Given two nodes x and y in a flow graph, x *dominates* y ($x \in dom(y)$) , if x appears in all paths from the *Start* node to y
- The node x *strictly dominates* y, if x dominates y and $x \neq y$
- x is the *immediate dominator* of y (denoted $idom(y)$), if x is the closest strict dominator of y
- A *dominator tree* shows all the immediate dominator relationships
- The *dominance frontier* of a node x, $DF(x)$, is the set of all nodes y such that
    - x dominates a predecessor of y ($p \in preds(y)$ *and* $x \in dom(p)$)
    - but x does not strictly dominate y ($x \notin dom(y) - \{y\}$)

Dominance frontiers - An Intuitive Explanation

- A definition in node *n* forces a ϕ-function in join nodes that lie just outside the region of the flow graph that *n* dominates; hence the name *dominance frontier*
- Informally, $DF(x)$ contains the *first* nodes reachable from *x* that *x* does not dominate, on *each* path leaving *x*
    - In example 1 (next slide), $DF(B1) = \emptyset$, since B1 dominates all nodes in the flow graph except *Start* and B1, and there is no path from B1 to *Start* or B1
    - In the same example, $DF(B2) = \{B2\}$, since B2 dominates all nodes except *Start*, B1, and B2, and there is a path from B2 to B2 (via the back edge)
    - Continuing in the same example, B5, B6, and B7 do not dominate any node and the first reachable nodes are B7, B7, and B2 (respectively). Therefore, $DF(B5) = DF(B6) = \{B7\}$ and $DF(B7) = \{B2\}$
    - In example 2 (second next slide), B5 dominates B6 and B7, but not B8; B8 is the first reachable node from B5 that B5 does not dominate; therefore, $DF(B5) = \{B8\}$

DF Example - 1

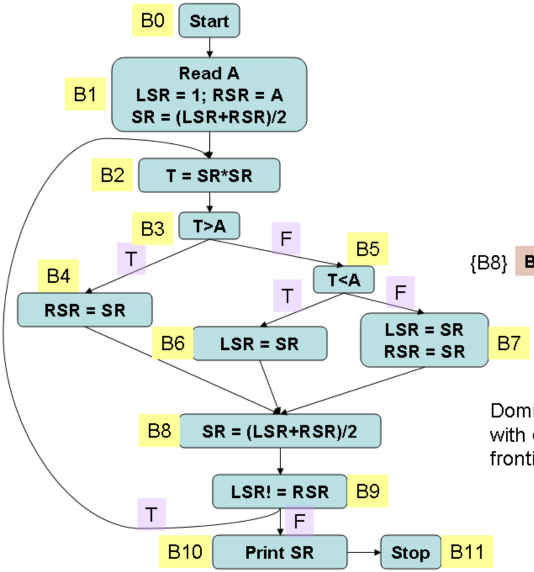


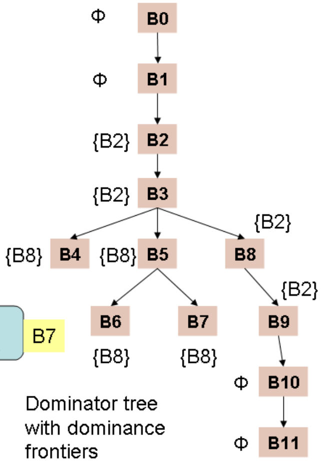


DF(x) is the set of all nodes y such that x dominates a predecessor of y, but x does not strictly dominate y

DF(x) contains the first nodes reachable from x, that x does not dominate

DF Example - 2





Dominator tree with dominance frontiers

Computation of Dominance Frontiers - 2

1. Identify each join node *x* in the flow graph
2. For each predecessor, *p* of *x* in the flow graph, traverse the dominator tree upwards from *p*, till *idom*(*x*)
3. During this traversal, add *x* to the *DF*-set of each node met

- In example 1 (second previous slide), consider the join node B2; its predecessors are B1 and B7
  - B1 is also *idom*(*B*2) and hence is not considered
  - Starting from B7 in the dominator tree, in the upward traversal till B1 (i.e., *idom*(*B*2)) B2 is added to the *DF* sets of B7, B3, and B2
- In example 2 (previous slide), consider the join node B8; its predecessors are B4, B6, and B7
  - Consider B4: B8 is added to *DF*(*B*4)
  - Consider B6: B8 is added to *DF*(*B*6) and *DF*(*B*5)
  - Consider B7: B8 is added to *DF*(*B*7); B8 has already been added to *DF*(*B*5)
  - All the above traversals stop at B3, which is *idom*(*B*8)

DF Algorithm

```
{
  for all nodes $n$ in the flow graph do
  $DF(n) = \emptyset$;
  for all nodes $n$ in the flow graph do {
  /* It is enough to consider only join nodes */
  /* Other nodes automatically get their DF sets */
  /* computed during this process /*
    for each predecessor $p$ of $n$ in the flow graph do {
        $t = p$;
        while ($t \neq idom(n)$) do {
            $DF(t) = DF(t) \cup \{n\}$;
            $t = idom(t)$;
        }
     }
  }
}
```

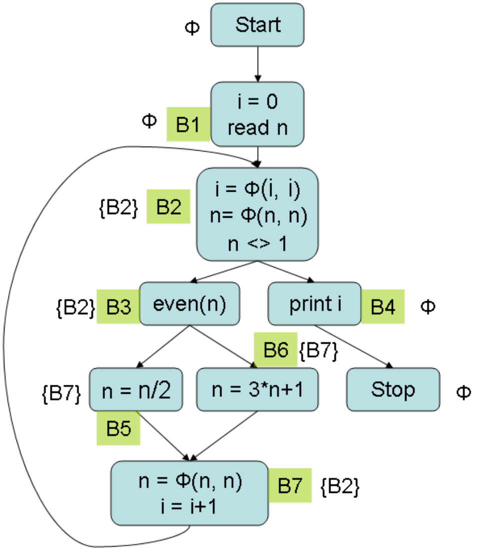
Minimal SSA Form Construction 1

1. Compute *DF* sets for each node of the flow graph
2. For each variable *v*, place trivial ϕ-functions in the nodes of the flow graph using the algorithm *place-phi-function(v)*
3. Rename variables using the algorithm *Rename-variables(x,B)*

ϕ-Placement Algorithm

- The ϕ-placement algorithm picks the nodes n_i with assignments to a variable
- It places trivial ϕ-functions in all the nodes which are in $DF(n_i)$, for each i
- It uses a work list (i.e., queue) for this purpose

ϕ-function placement Example



Dominance frontier is written beside BB no.

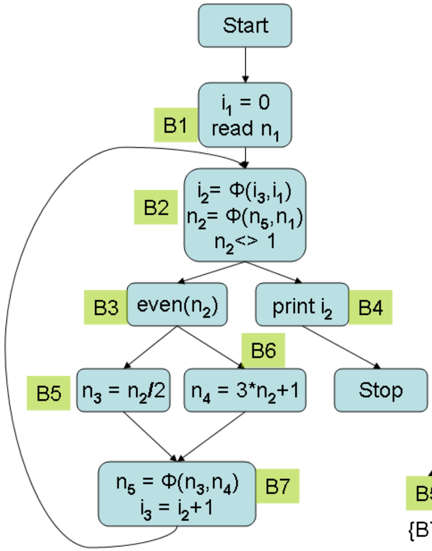
The function *place-phi-function(v)* - 1

```
function Place-phi-function(v) // v is a variable
// This function is executed once for each variable in the flow graph
begin
  // has-phi(B, v) is true if a $\phi$-function has already
  // been placed in B, for the variable v
  // processed(B) is true if B has already been processed once
  // for variable v
  for all nodes B in the flow graph do
     has-phi(B, v) = false; processed(B) = false;
  end for
  W = $\emptyset$; // W is the work list
  // Assignment-nodes(v) is the set of nodes containing
  // statements assigning to v
  for all nodes B $\in$ Assignment-nodes(v) do
     processed(B) = true; Add(W, B);
  end for
```

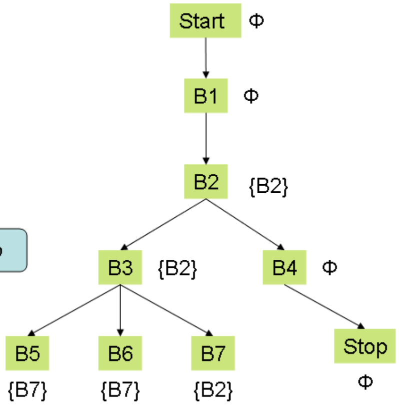
The function *place-phi-function(v)* - 2

```
while $W \neq \emptyset$ do
begin
  $B$ = $Remove(W)$;
  for all nodes $y \in DF(B)$ do
    if (not $has\text{-}phi(y, v)$) then
    begin
      place $< v = \phi(v, v, ..., v) >$ in $y$;
      $has\text{-}phi(y, v)$ = $true$;
      if (not $processed(y)$) then
      begin $processed(y)$ = $true$;
            $Add(W, y)$;
      end
    end
  end for
end
end
```

SSA Form Construction Example - 1

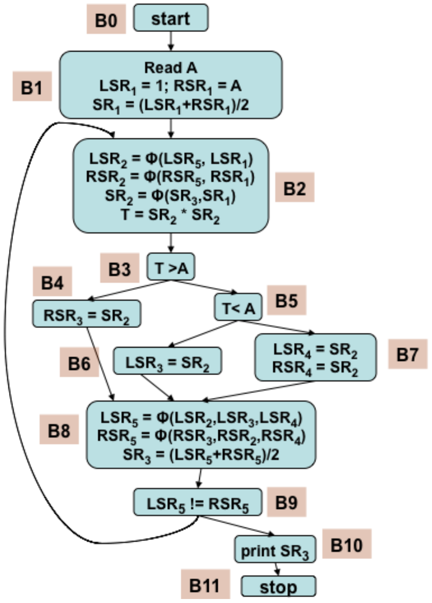


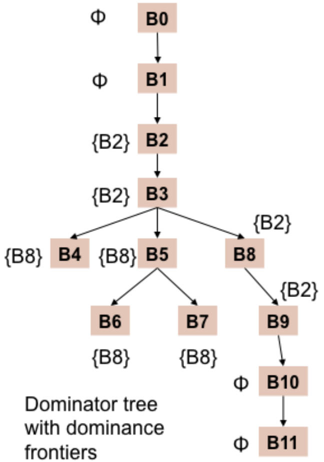
SSA form



Dominator tree with dominance frontier

SSA Form Construction Example - 2





Dominator tree with dominance frontiers

Minimal SSA Form Construction 2

Renaming Algorithm

- The renaming algorithm performs a top-down traversal of the dominator tree
- A separate pair of version stack and version counter are used for each variable
    - The top element of the version stack V is always the version to be used for a variable usage encountered (in the appropriate range, of course)
    - The counter v is used to generate a new version number
- The alogorithm shown later is for a single variable only; a similar algorithm is executed for all variables with an array of version stacks and counters

The Renaming Algorithm

- An SSA form should satisfy the *dominance property*:
  - the definition of a variable dominates each use or
  - when the use is in a ϕ-function, the predecessor of the use
- Therefore, it is apt that the renaming algorithm performs a top-down traversal of the dominator tree
  - Renaming for non-ϕ-statements is carried out while visiting a node *n*
  - Renaming parameters of a ϕ-statement in a node *n* is carried out while visiting the appropriate predecessors of *n*

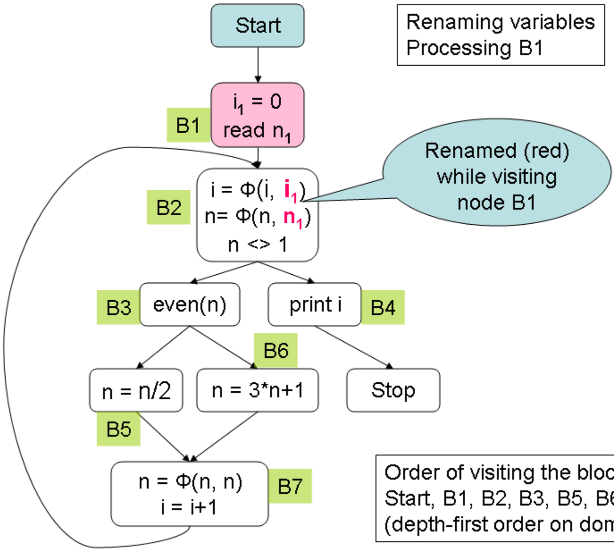
The function *Rename-variables(x,B)*

```
function Rename-variables($x, B$) // $x$ is a variable and $B$ is a block
begin
  $v_e$ = Top($V$); // $V$ is the version stack of $x$
  // variables are defined before use; hence no renaming can
  // happen on empty stack
  for all statements $s \in B$ do
    if $s$ is a non-$\phi$ statement then
        replace all uses of $x$ in the RHS($s$) with Top($V$);
    if $s$ defines $x$ then
    begin
      replace $x$ with $x_v$ in its definition; push $x_v$ onto $V$;
      // $x_v$ is the renamed version of $x$ in this definition
      $v = v + 1$; // $v$ is the version number counter
    end
  end for
```

The function *Rename-variables(x,B)*

```
  for all successors $s$ of $B$ in the flow graph do
    $j$ = predecessor index of $B$ with respect to $s$
    for all $\phi$-functions $f$ in $s$ which define $x$ do
       replace the $j^{th}$ operand of $f$ with $Top(V)$;
    end for
  end for
  for all children $c$ of $B$ in the dominator tree do
     $Rename\text{-}variables(x, c)$;
  end for
  repeat $Pop(V)$; until ($Top(V) == v_e$);
end
begin // calling program
  for all variables $x$ in the flow graph do
     $V = \emptyset$; $v$ = 1; push 0 onto $V$; // end-of-stack marker
     $Rename\text{-}variables(x, Start)$;
  end for
end
```

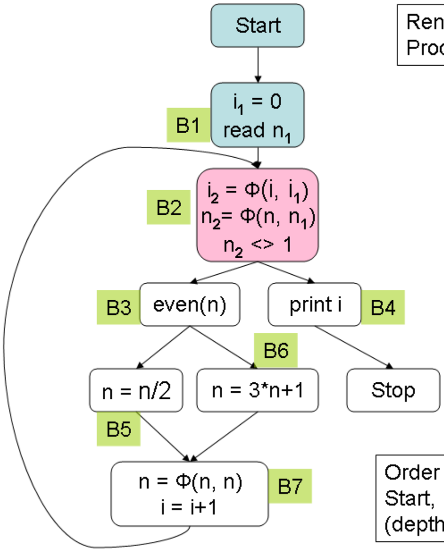
Renaming Variables Example 0.1



Renaming variables
Processing B1

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

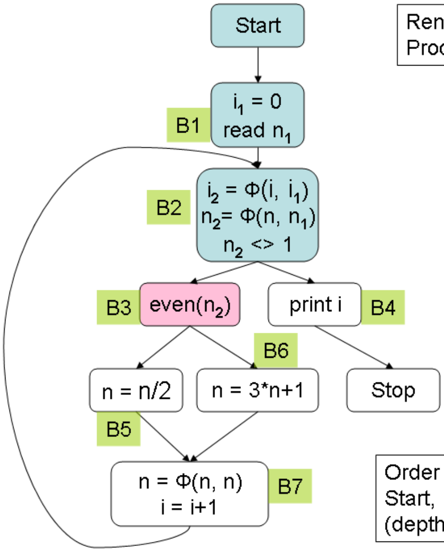
Renaming Variables Example 0.2



Renaming variables
Processing B2

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

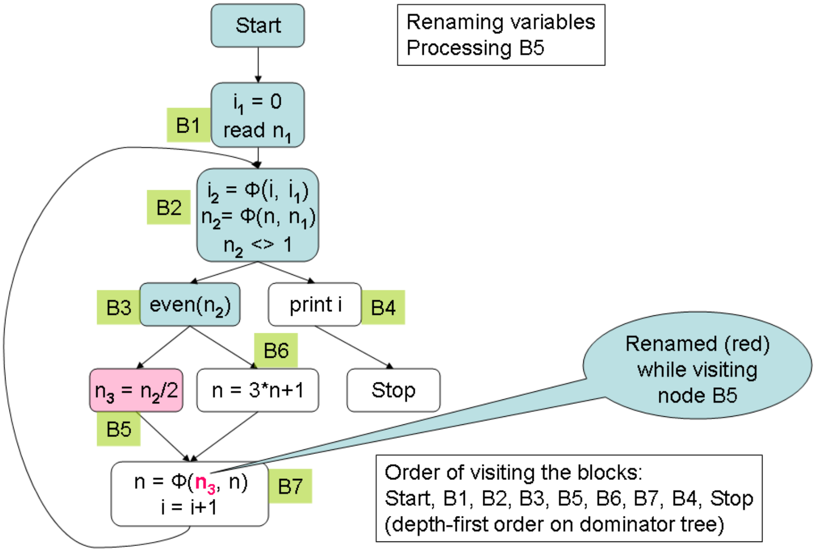
Renaming Variables Example 0.3



Renaming variables
Processing B3

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

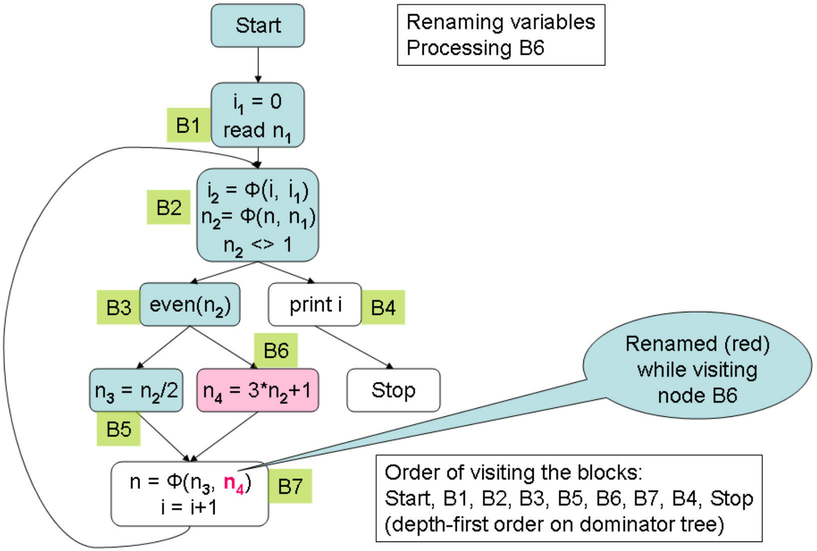
Renaming Variables Example 0.4



Renaming variables
Processing B5

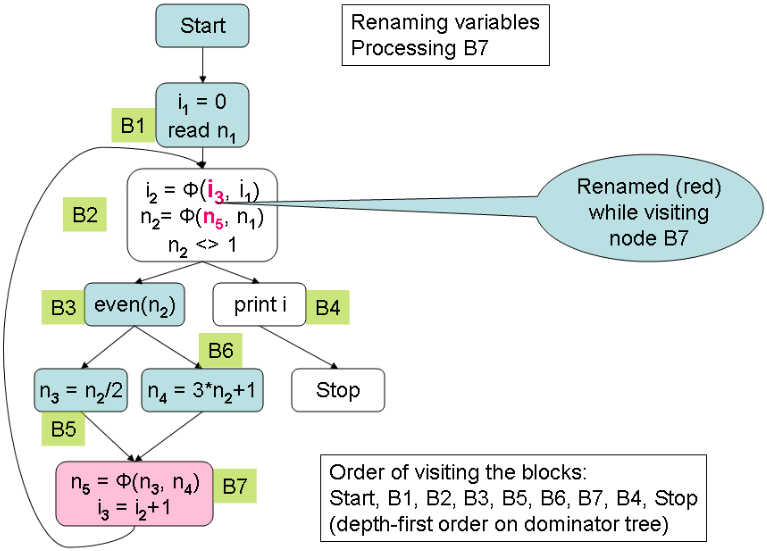
Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

Renaming Variables Example 0.5



Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

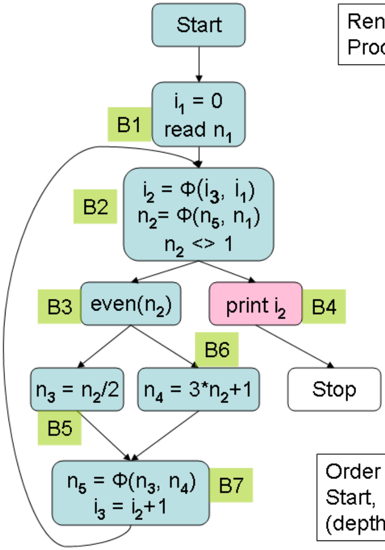
Renaming Variables Example 0.6



Renaming variables
Processing B7

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

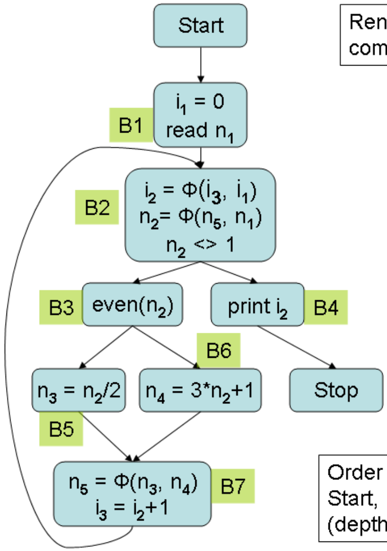
Renaming Variables Example 0.7



Renaming variables
Processing B4

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

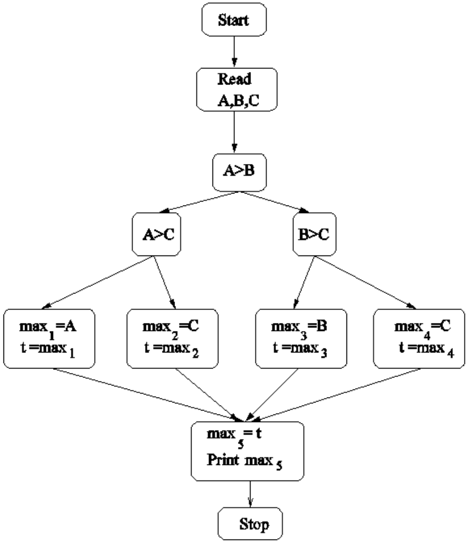
Renaming Variables Example 0.8



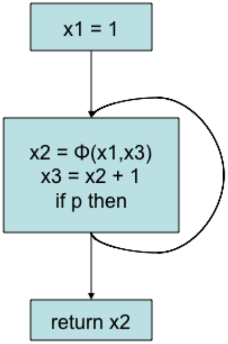
Renaming variables completed

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

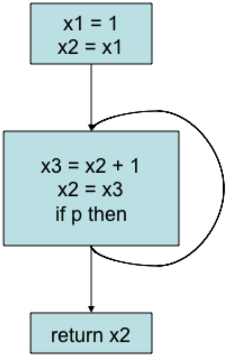
Translation to Machine Code - 1



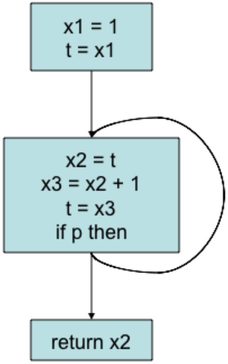
Translation to Machine Code - 2



Original program



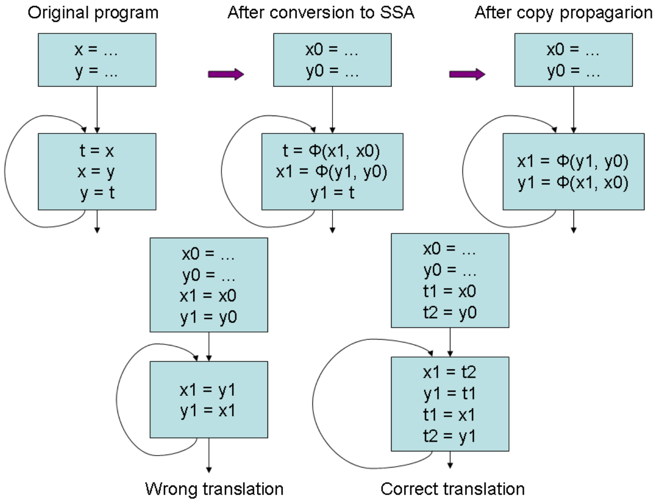
Wrong translation returned value is incorrect



Correct translation

Translation to Machine Code - 3

The parameters of all ϕ-functions in a basic block are supposed to be read concurrently before any other evaluation begins



Optimization Algorithms with SSA Forms

- Dead-code elimination
    - Very simple, since there is exactly one definition reaching each use
    - Examine the *du-chain* of each variable to see if its use list is empty
    - Remove such variables and their definition statements
    - If a statement such as $x = y + z$ (or $x = \phi(y_1, y_2)$) is deleted, care must be taken to remove the deleted statement from the *du-chains* of y and z (or y_1 and y_2)
- Simple constant propagation
- Copy propagation
- Conditional constant propagation and constant folding
- Global value numbering

Simple Constant Propagation

```
{   Stmtpile = {S|S is a statement in the program}
    while Stmtpile is not empty {
        S = remove(Stmtpile);
        if S is of the form $x = \phi(c, c, ..., c)$ for some constant $c$
               replace S by $x = c$
        if S is of the form $x = c$ for some constant $c$
             delete S from the program
             for all statements T in the du-chain of $x$ do
                  substitute $c$ for $x$ in T; simplify T
                  Stmtpile = Stmtpile $\cup$ {T}
}
```

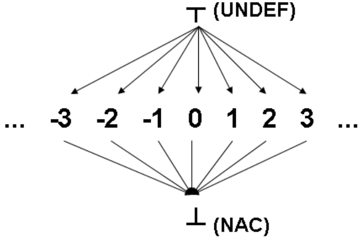
Copy propagation is similar to constant propagation

- A single-argument ϕ-function, $x = \phi(y)$, or a copy statement, $x = y$ can be deleted and y substituted for every use of x

The Constant Propagation Framework - An Overview

$m(y)$	$m(z)$	$m'(x)$
UNDEF	UNDEF	UNDEF
	c_2	UNDEF
	NAC	NAC
c_1	UNDEF	UNDEF
	c_2	$c_1 + c_2$
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC

$any \sqcap UNDEF = any$
$any \sqcap NAC = NAC$
$c_1 \sqcap c_2 = NAC, \text{ if } c_1 \neq c_2$
$c_1 \sqcap c_2 = c_1, \text{ if } c_1 = c_2$



Conditional Constant Propagation - 1

- SSA forms along with extra edges corresponding to *d-u* information are used here
  - Edge from every definition to each of its uses in the SSA form (called henceforth as *SSA edges*)
- Uses both flow graph and SSA edges and maintains two different work-lists, one for each (*Flowpile* and *SSApile* , resp.)
- Flow graph edges are used to keep track of reachable code and SSA edges help in propagation of values
- Flow graph edges are added to *Flowpile*, whenever a branch node is symbolically executed or whenever an assignment node has a single successor

Conditional Constant Propagation - 2

- SSA edges coming out of a node are added to the SSA work-list whenever there is a change in the value of the assigned variable at the node
- This ensures that all *uses* of a definition are processed whenever a definition changes its lattice value.
- This algorithm needs only one lattice cell per *variable* (globally, not on a per node basis) and two lattice cells per node to store expression values
- Conditional expressions at branch nodes are evaluated and depending on the value, either one of outgoing edges (corresponding to *true* or *false*) or both edges (corresponding to $\perp$) are added to the worklist
- However, at any join node, the *meet* operation considers only those predecessors which are marked *executable*.

CCP Algorithm - Contd.

```
// $\mathcal{G} = (\mathcal{N}, \mathcal{E}_f, \mathcal{E}_s)$ is the SSA graph,
// with flow edges and SSA edges, and
// $\mathcal{V}$ is the set of variables used in the SSA graph
begin
  $Flowpile = \{(Start \rightarrow n) \mid (Start \rightarrow n) \in \mathcal{E}_f\}$;
  $SSApile = \emptyset$;
  for all $e \in \mathcal{E}_f$ do $e.executable = false$; end for
  //$v.cell$ is the lattice cell associated with the variable $v$
  for all $v \in \mathcal{V}$ do $v.cell = \top$; end for
  // $y.oldval$ and $y.newval$ store the lattice values
  // of expressions at node $y$
  for all $y \in \mathcal{N}$ do
    $y.oldval = \top$; $y.newval = \top$;
  end for
```

CCP Algorithm - Contd.

```
while ($Flowpile \neq \emptyset$) or ($SSApile \neq \emptyset$) do
begin
  if ($Flowpile \neq \emptyset$) then
  begin
    $(x, y)$ = remove($Flowpile$);
    if (not $(x, y).executable$) then
    begin
      $(x, y).executable$ = true;
      if ($\phi$-$present(y)$) then $visit$-$\phi(y)$
        else if ($first$-$time$-$visit(y)$) then $visit$-$expr(y)$;
      // $visit$-$expr$ is called on $y$ only on the first visit
      // to $y$ through a flow edge; subsequently, it is called
      // on $y$ on visits through SSA edges only
      if ($flow$-$outdegree(y) == 1$) then
        // Only one successor flow edge for y
        $Flowpile = Flowpile \cup \{(y, z) \mid (y, z) \in \mathcal{E}_f\}$;
    end
```

CCP Algorithm - Contd.

```
            // if the edge is already marked, then do nothing
        end
        if ($SSApile \neq \emptyset$) then
          begin
            $(x, y)$ = remove($SSApile$);
            if ($\phi$-present($y$)) then visit-$\phi$($y$)
              else if (already-visited($y$)) then visit-expr($y$);
              // A false returned by already-visited implies
              // that $y$ is not yet reachable through flow edges
          end
        end // Both piles are empty
    end
    function $\phi$-present($y$) // $y \in \mathcal{N}$
    begin
      if $y$ is a $\phi$-node then return true
        else return false
    end
```

CCP Algorithm - Contd.

```
function $visit\text{-}\phi(y)$ // $y \in \mathcal{N}$
begin
  $y.newval = \top$; //$\| y.instruction.inputs \|$ is the number of
  // parameters of the $\phi$-instruction at node $y$
  for $i = 1$ to $\| y.instruction.inputs \|$ do
    Let $p_i$ be the $i^{th}$ predecessor of $y$ ;
    if $((p_i, y).executable)$ then
    begin
      Let $a_i = y.instruction.inputs[i]$;
      // $a_i$ is the $i^{th}$ input and $a_i.cell$ is the lattice cell
      // associated with that variable
      $y.newval = y.newval \sqcap a_i.cell$;
    end
  end for
```

CCP Algorithm - Contd.

```
    if ($y.newval < y.instruction.output.cell$) then
    begin
        $y.instruction.output.cell = y.newval$;
        $SSApile = SSApile \cup \{(y, z) \mid (y, z) \in \mathcal{E}_s\}$;
    end
end

function $already\text{-}visited(y)$ // $y \in \mathcal{N}$
// This function is called when processing an SSA edge
begin // Check in-coming flow graph edges of $y$
    for all $e \in \{(x, y) \mid (x, y) \in \mathcal{E}_f\}$
        if $e.executable$ is true for at least one edge $e$
            then return $true$ else return $false$
    end for
end
```

CCP Algorithm - Contd.

```
function first-time-visit($y$) // $y \in \mathcal{N}$
// This function is called when processing a flow graph edge
begin // Check in-coming flow graph edges of $y$
    for all $e \in \{(x, y) \mid (x, y) \in \mathcal{E}_f\}$
        if $e.executable$ is true for more than one edge $e$
            then return false else return true
    end for
    // At least one in-coming edge will have executable true
    // because the edge through which node $y$ is entered is
    // marked as executable before calling this function
end
```

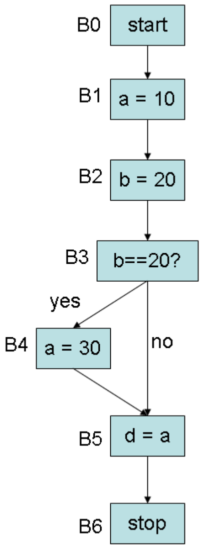
CCP Algorithm - Contd.

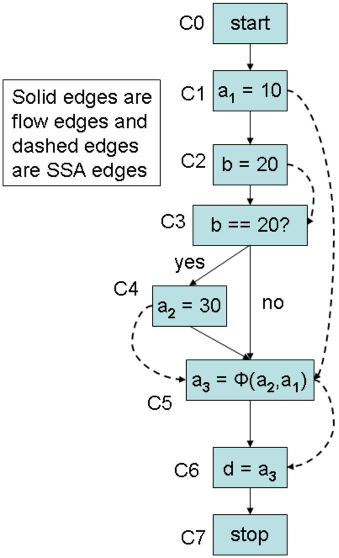
```
function $visit\text{-}expr(y)$ // $y \in \mathcal{N}$
begin
  Let $input_1 = y.instruction.inputs[1]$;
  Let $input_2 = y.instruction.inputs[2]$;
  if ($input_1.cell == \perp$ or $input_2.cell == \perp$) then
    $y.newval = \perp$
  else if ($input_1.cell == \top$ or $input_2.cell == \top$) then
        $y.newval = \top$
      else // evaluate expression at $y$ as per lattice evaluation rules
        $y.newval = evaluate(y)$;
        // It is easy to handle instructions with one operand
  if $y$ is an assignment node then
    if ($y.newval < y.instruction.output.cell$) then
    begin
      $y.instruction.output.cell = y.newval$;
      $SSApile = SSApile \cup \{(y, z) \mid (y, z) \in \mathcal{E}_s\}$;
    end
```

CCP Algorithm - Contd.

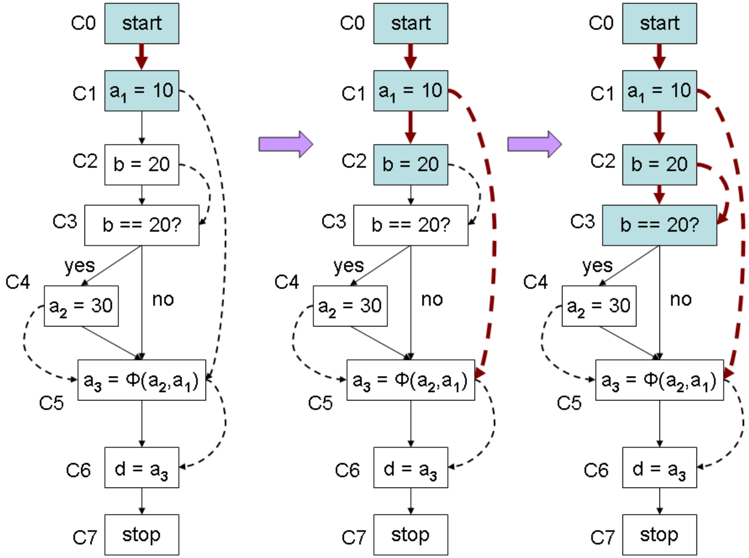
```
    else if $y$ is a branch node then
      begin
        if ($y.newval < y.oldval$) then
        begin
          $y.oldval = y.newval$;
          switch($y.newval$)
            case $\perp$: // Both true and false branches are equally likely
              $Flowpile = Flowpile \cup \{(y, z) \mid (y, z) \in \mathcal{E}_f\}$;
            case true: $Flowpile = Flowpile \cup \{(y, z) \mid (y, z) \in \mathcal{E}_f$ and
                                   $(y, z)$ is the true branch edge at $y\}$;
            case false: $Flowpile = Flowpile \cup \{(y, z) \mid (y, z) \in \mathcal{E}_f$ and
                                   $(y, z)$ is the false branch edge at $y\}$;
          end switch
        end
      end
  end
```

CCP Algorithm - Example - 1

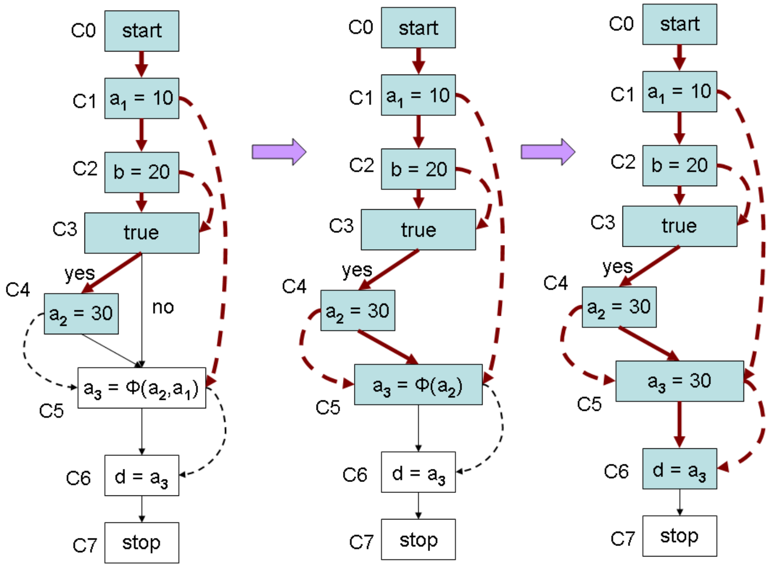




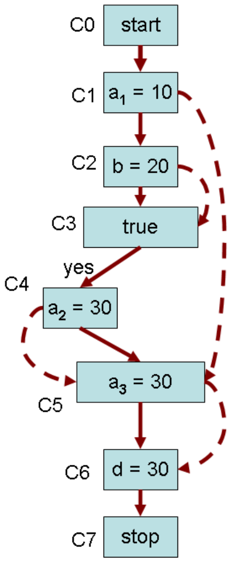
CCP Algorithm - Example 1 - Trace 1



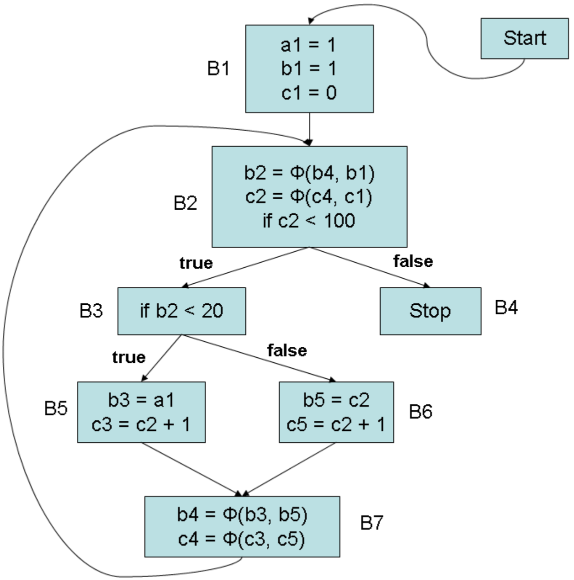
CCP Algorithm - Example 1 - Trace 2



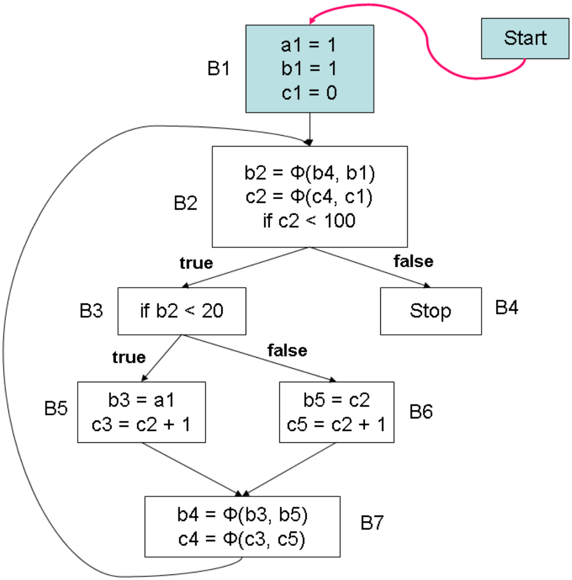
CCP Algorithm - Example 1 - Trace 3



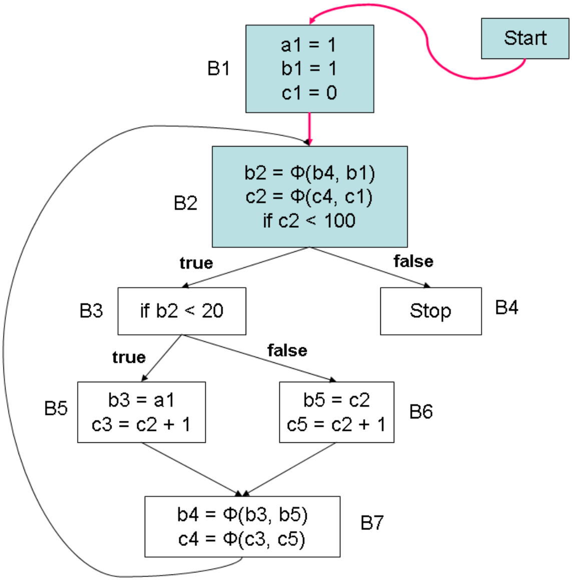
CCP Algorithm - Example 2



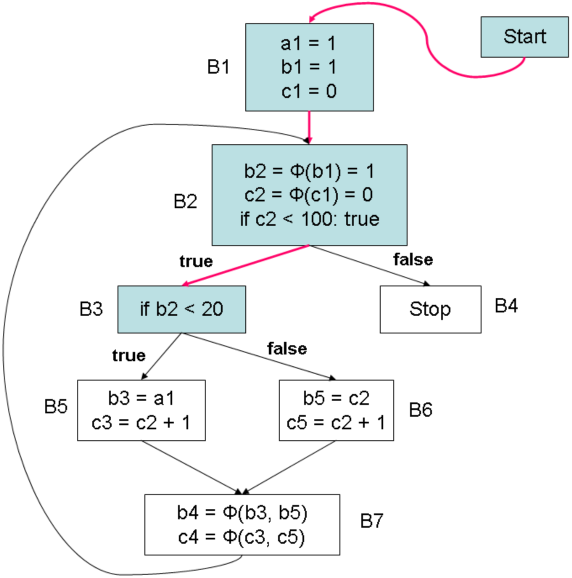
CCP Algorithm - Example 2 - Trace 1



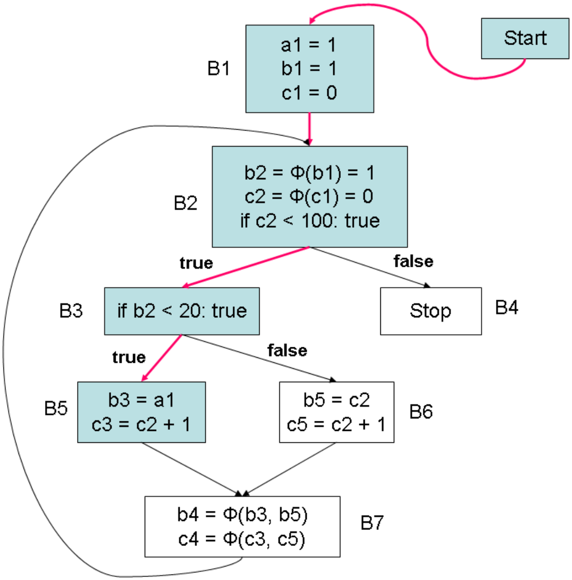
CCP Algorithm - Example 2 - Trace 2



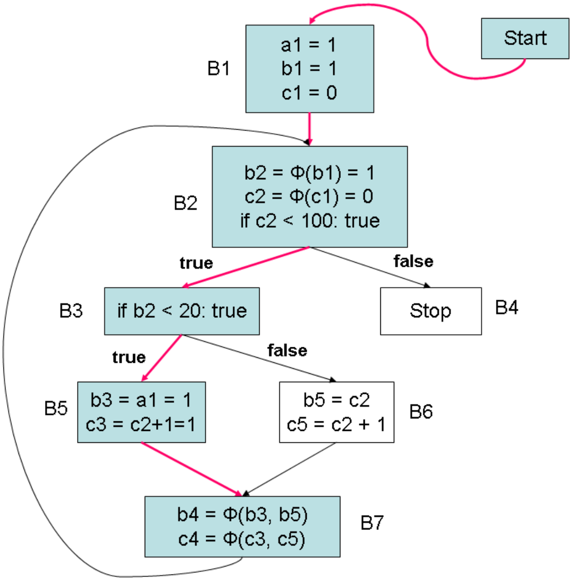
CCP Algorithm - Example 2 - Trace 3



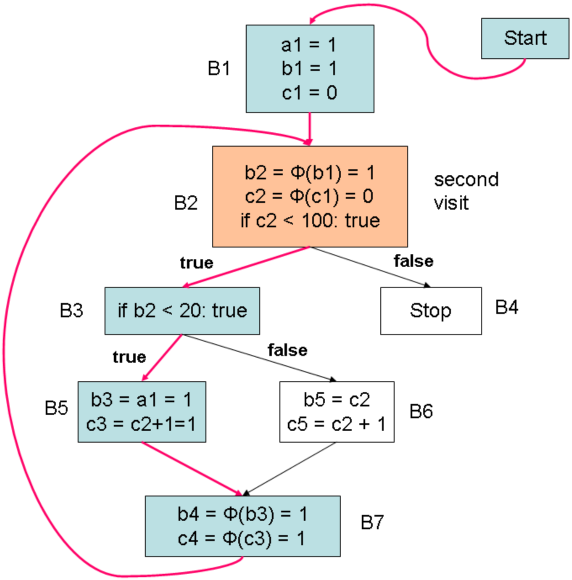
CCP Algorithm - Example 2 - Trace 4



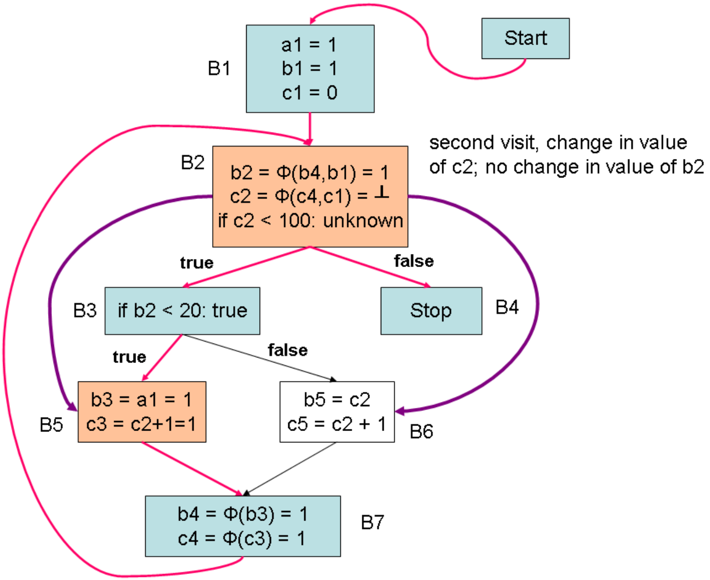
CCP Algorithm - Example 2 - Trace 5



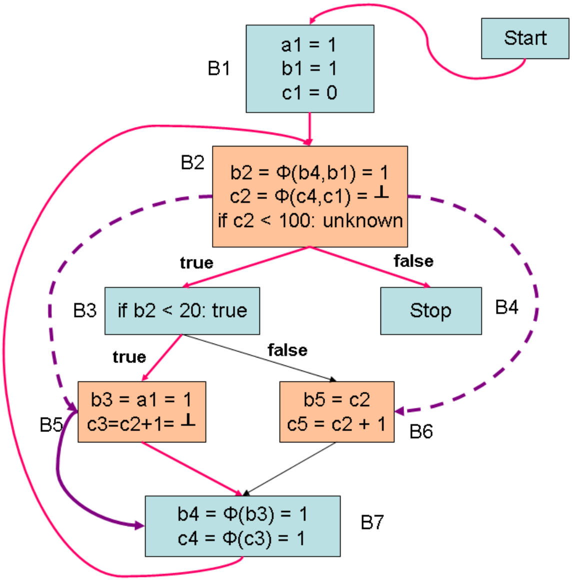
CCP Algorithm - Example 2 - Trace 6



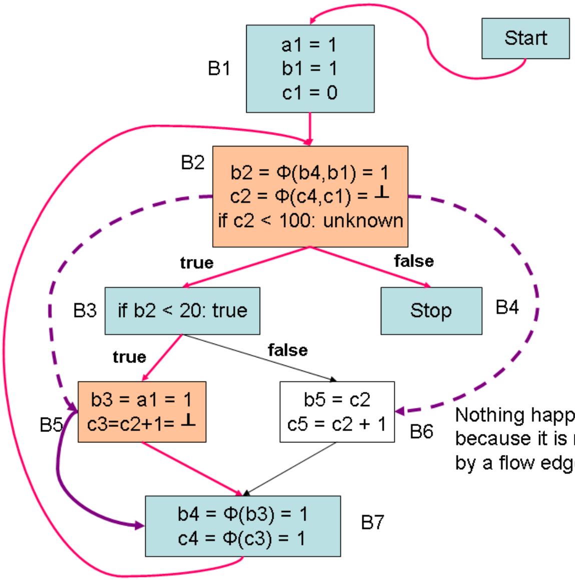
CCP Algorithm - Example 2 - Trace 7



CCP Algorithm - Example 2 - Trace 8

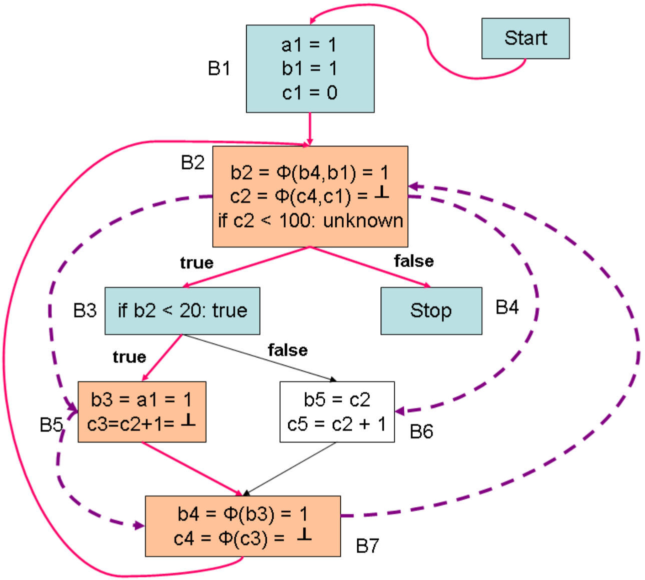


CCP Algorithm - Example 2 - Trace 9

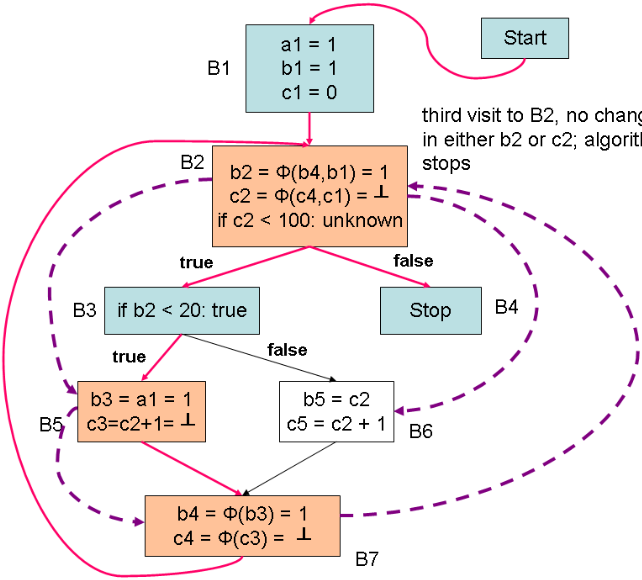


Nothing happens in B6 because it is not reachable by a flow edge

CCP Algorithm - Example 2 - Trace 10



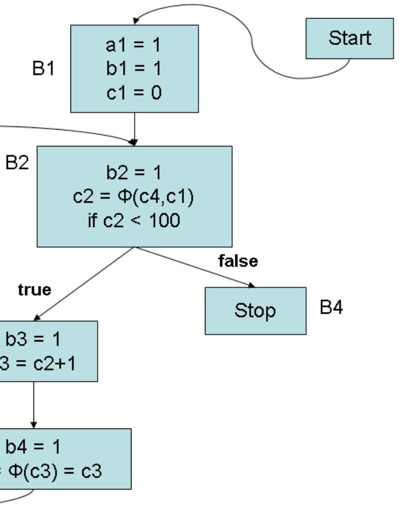
CCP Algorithm - Example 2 - Trace 11



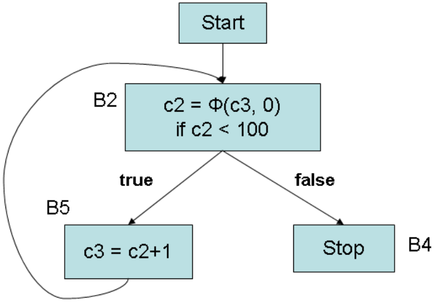
third visit to B2, no change in either b2 or c2; algorithm stops

CCP Algorithm - Example 2 - Trace 12

After first round of simplification



CCP Algorithm - Example 2 - Trace 13



After second round of simplification –
elimination of dead code, elimination
of trivial Φ-functions, copy propagation etc.

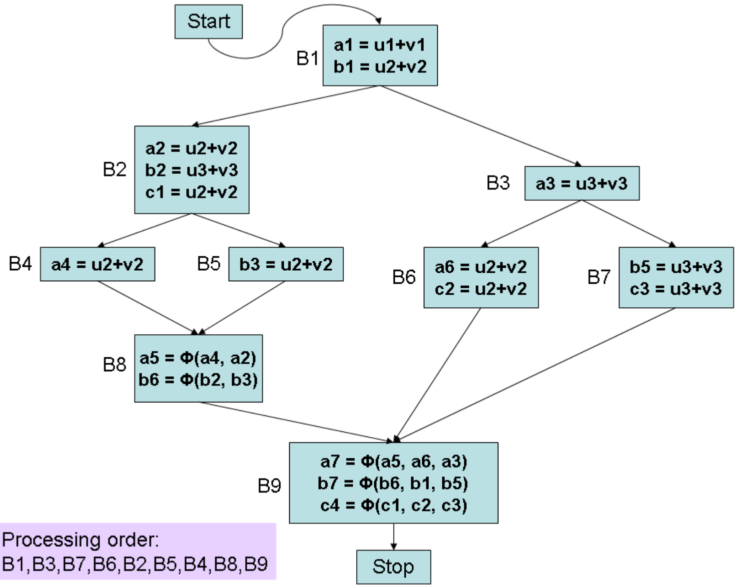
Value Numbering with SSA Forms

- Global value numbering scheme
  - Similar to the scheme with extended basic blocks
  - Scope of the tables is over the dominator tree
  - Therefore more redundancies can be caught
    - For example, an assignment $a_{10} = u_1 + v_1$ in block $B9$ (if present) can use the value of the expression $u_1 + v_1$ of block $B1$, since $B1$ is a dominator of $B9$
- No *d-u* or *u-d* edges needed
- Uses *reverse post order* on the DFS tree of the SSA graph to process the dominator tree
  - This ensures that definitions are processed before use
- Back edges make the algorithm find *fewer* equivalences (more on this later)

Value Numbering with SSA Forms

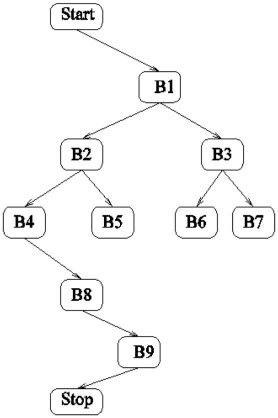
- Variable names are not reused in SSA forms
  - Hence, no need to restore old entries in the scoped *HashTable* when the processing of a block is completed
  - Just deleting new entries will be sufficient
- Any copies generated because of common subexpressions can be deleted immediately
- Copy propagation is carried out during value-numbering
- Ex: Copy statements generated due to value numbering in blocks B2, B4, B5, B6, B7, and B8 can be deleted
- The *ValnumTable* stores the SSA name and its value number and is global; it is not scoped over the dominator tree (reasons in the next slide)
- Value numbering transformation retains the *dominance property* of the SSA form
  - Every definition dominates all its uses or predecessors of uses (in case of *phi*-functions)

Example: An SSA Form

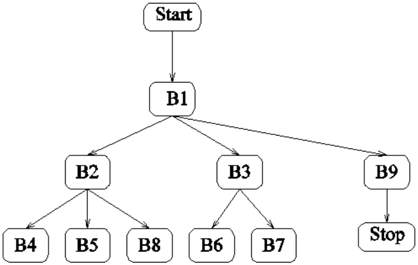


Processing order:
B1,B3,B7,B6,B2,B5,B4,B8,B9

Dominator Tree and Reverse Post order



Postorder on the DFS tree:
Stop, B9, B8, B4, B5, B2, B6, B7, B3, B1, Start



Reverse postorder on the SSA graph that is used with the dominator tree above:

Start,B1,B3,B7,B6,B2,B5,B4,B8,B9,Stop

References

1. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck
   Efficiently Computing Static Single Assignment Form and the Control Dependence Graph
   ACM Trans. Program. Lang. Syst. 13(4): 451-490 (1991)
2. Mark N. Wegman, F. Kenneth Zadeck
   Constant Propagation with Conditional Branches
   ACM Trans. Program. Lang. Syst. 13(2): 181-210 (1991)

Outline

Outline

Iteration Spaces and Dependences

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
      A[t+1][i] = f(A[t][i+1], A[t][i], A[t][i-1]);
```

1. **Iteration Domains**
   - Every statement has a domain or an index **set –** instances that have to be executed
   - Each instance is a vector (of loop index values from outermost to innermost) $D_S = \{[t, i] \mid 0 \le t \le T - 1,\ 1 \le i \le N\}$
2. **Dependences**
   - A dependence is a **relation** between domain instances that are in conflict (more on next slide)

Lexicographic Ordering

- **Lexicographic ordering**: $\succ$, $\prec$, $\vec{x} \succ \vec{y}$, $\succ \vec{0}$
- **Transformations** as a way to provide multi-dimensional timestamps
- Code generation: **Scanning points in the transformed space in lexicographically increasing order**

Domains, Dependences, and Transformations

```
for (i=1; i<=N-1; i++)
  for (j=1; j<=N-1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);
```

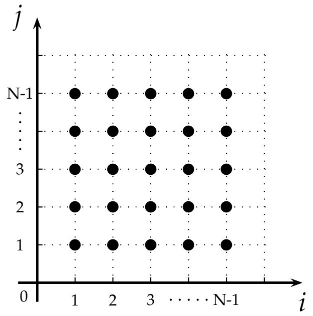


Figure: Original space (i, j)

- **Domain**: $\{[i, j] \mid 1 \leq i, j \leq N - 1\}$

Domains, Dependences, and Transformations

```
for (i=1; i<=N-1; i++)
  for (j=1; j<=N-1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);
```

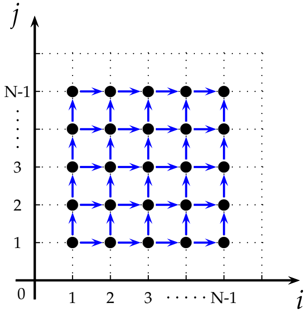


Figure: Original space (i, j)

- **Dependences**:
  1. $\{[i,j] \rightarrow [i+1,j] \mid 1 \leq i \leq N-2, 0 \leq j \leq N-1\}$ — **(1,0)**
  2. $\{[i,j] \rightarrow [i,j+1] \mid 1 \leq i \leq N-1, 0 \leq j \leq N-2\}$ — **(0,1)**

Domains, Dependences, and Transformations

```
for (i=1; i<=N-1; i++)
  for (j=1; j<=N-1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);
```

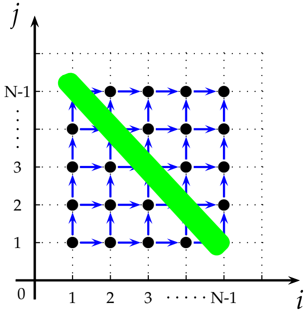


Figure: Original space (i, j)

- **Dependences**:
  1. $\{[i,j] \to [i+1,j] \mid 1 \le i \le N-2, 0 \le j \le N-1\}$ — **(1,0)**
  2. $\{[i,j] \to [i,j+1] \mid 1 \le i \le N-1, 0 \le j \le N-2\}$ — **(0,1)**

Domains, Dependences, and Transformations

```
for (i=1; i<=N-1; i++)
  for (j=1; j<=N-1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);
```

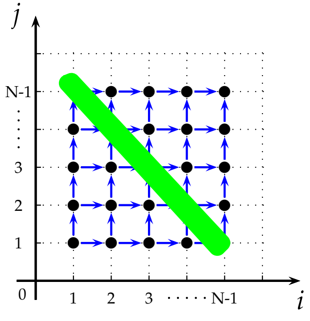


Figure: Original space (i, j)

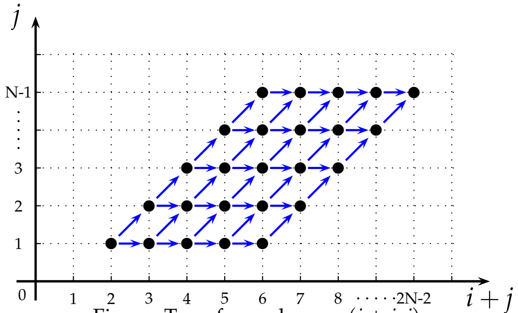


Figure: Transformed space $(i + j, j)$

- **Transformation**: $T(i, j) = (i + j, j)$
  - Dependences: (1,0) and (0,1) now become (1,0) and (1,1) resp.
  - Inner loop is now parallel

Domains, Dependences, and Transformations

```
for (i=1; i<=N-1; i++)
  for (j=1; j<=N-1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);
```

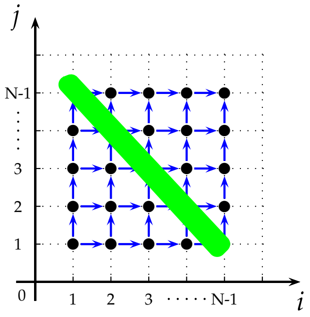


Figure: Original space (i, j)

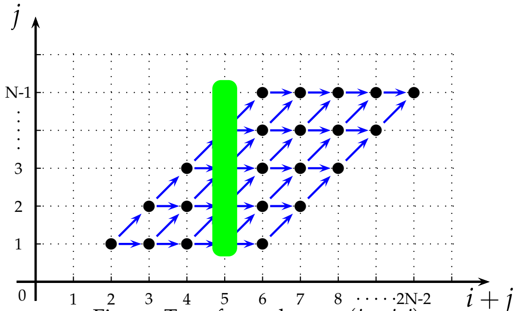


Figure: Transformed space $(i + j, j)$

- **Transformation**: $T(i, j) = (i + j, j)$
  - Dependences: (1,0) and (0,1) now become (1,0) and (1,1) resp.
  - Inner loop is now parallel

Dependences: Another Example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    A[i] = f(A[i+1], A[i], A[i-1]);
```

- Compute the dependences
- Transitivity in dependences?
- Remove transitively covered dependences.

DEPENDENCES: ANOTHER EXAMPLE

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    A[i] = f(A[i+1], A[i], A[i-1]);
```

- Compute the dependences
- Transitivity in dependences?
- Remove transitively covered dependences.

Dependences: Yet Another Example

```
for (i = 0; i < N; i++)
  for (j = 1; j < i; j++)
    A[j] = A[j] - A[j]/A[i];
```

- Compute the dependences.

Dependence Representations

1. Distance vectors: constant dependences
2. Dependence levels: depth at which a dependence is carried
3. Direction vectors: direction of the dependence along each dimension
4. Dependence as presburger formulae, relations on integer sets with affine constraints and existential quantifiers

Dependence Testing

- GCD test, GCD tightening of constraints
- Guassian elimination, Fourier-Motzkin elimination (super-exponential) complexity
- Omega test

Outline

CHARACTERIZING REUSE

- Reuse through multi-dimensional array accesses
  1. Self reuse
  2. Group reuse
- In space or in time?
  1. Spatial reuse (self or group)
  2. Temporal reuse (self or group)
- Under what conditions does an access exhibit spatial or temporal reuse along a specific outer loop?
  - This topic is well-covered in the Dragon textbook.
- Degree of temporal reuse: Dimensionality of the iteration space minus rank of the access function
  Eg: *for* (i, j, k), access A[i + j][j][j] has an access function of rank two in an iteration space of dimensionality three $\rightarrow$ one degree of temporary reuse.

Representation of Array Accesses

1. Linear Algebraic representation of “regular” accesses
2. Affine access functions can be analyzed by the compiler easily for reuse, dependences, optimization, and parallelization
3. Refer to the definition of affine functions earlier
4. Handling compositions of mod and floordiv functions in accesses requires additional techniques to determine spatial and temporal reuse

Loop Nests: Some definitions

- **Perfectly nested** loop nest: A sequence of successively nested loops (from outermost to innermost) where every loop other than the innermost one has a single loop as the only statement in its body.
- **Imperfectly nested**: not perfectly nests.

```
// Perfectly nested.
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      S(t, i, j);
```

```
// (t, i, j) is imperfectly nested, but
// (t, i) is perfectly nested.
for (t = 0; t < T; t++) {
  for (i = 1; i < N+1; i++) {
    S1(t, i);
    for (j = 1; j < N+1; j++)
      S2(t, i, j);
  }
}
```

Outline

AFFINE TRANSFORMATIONS

- Examples of affine functions of i, j: $i+j, i-j, i+1, 2i+5$
- Not affine: $ij, i^2, i^2+j^2, a[j]$

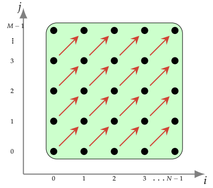


Figure: Iteration space

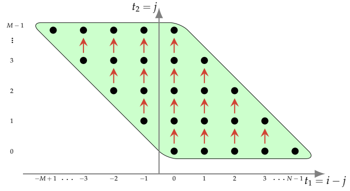


Figure: Transformed space

```
// O(N) synchronization if j is parallelized.
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
     A[i+1][j+1] = f(A[i][j]);
```

```
// Synchronization-free.
#pragma omp parallel for private(t2)
for (t1=-M+1; t1<=N-1; t1++)
  for (t2 = max(0,-t1); t2 <= min(M-1,N-1-t1); t2++)
    A[t1+t2+1][t2+1] = f(A[t1+t2][t2]);
```

- Transformation: $(i,j) \rightarrow (i-j, j)$

Affine Transformations

- Examples of affine functions of i, j: $i + j$, $i - j$, $i + 1$, $2i + 5$
- Not affine: ij, i^2, $i^2 + j^2$, $a[j]$

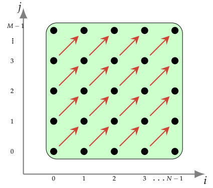


Figure: Iteration space

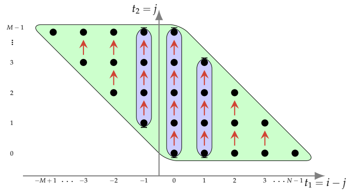


Figure: Transformed space

```
// O(N) synchronization if j is parallelized.
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
     A[i+1][j+1] = f(A[i][j]);
```

```
// Synchronization-free.
#pragma omp parallel for private(t2)
for (t1=-M+1; t1<=N-1; t1++)
  for (t2 = max(0,-t1); t2 <= min(M-1,N-1-t1); t2++)
    A[t1+t2+1][t2+1] = f(A[t1+t2][t2]);
```

- Transformation: $(i, j) \rightarrow (i - j, j)$

Affine Transformations

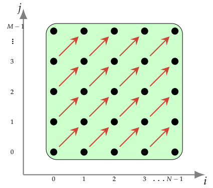


Figure: Iteration space

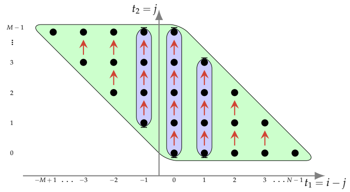


Figure: Transformed space

- Affine transformations are attractive because:
    - Preserve **collinearity** of points and **ratio of distances** between points
    - Code generation with affine transformations has thus been studied well (CLooG, ISL, OMEGA+)
    - Model a very rich class of loop re-orderings
    - Useful for several domains like dense linear algebra, stencil computations, image processing pipelines, deep learning

Finding Good Affine Transformations

(i, j)	Identity
(j, i)	Interchange
$(i + j, j)$	Skew i (by a factor of one w.r.t j)
$(i - j, -j)$	Reverse j and skew i
$(i, 2i + j)$	Skew j (by a factor of two w.r.t i)
$(2i, j)$	Scale i by a factor of two
$(i, j + 1)$	Shift j
$(i + j, i - j)$	More complex
$(i/32, j/32, i, j)$	Tile
$\ldots$	

- One-to-one functions
- Can be expressed using matrices: $T(i, j) = (i + j, j) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$.
- Unimodular and non-unimodular transformations

Finding Good Affine Transformations

(i, j)	Identity
(j, i)	Interchange
$(i + j, j)$	Skew i (by a factor of one w.r.t j)
$(i - j, -j)$	Reverse j and skew i
$(i, 2i + j)$	Skew j (by a factor of two w.r.t i)
$(2i, j)$	Scale i by a factor of two
$(i, j + 1)$	Shift j
$(i + j, i - j)$	More complex
$(i/32, j/32, i, j)$	Tile
$\ldots$	

- One-to-one functions
- Can be expressed using matrices: $T(i,j) = (i+j, j) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$.
- Unimodular and non-unimodular transformations

Dependences

- Dependences are determined pairwise between conflicting accesses

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                           A[t%2][i][j+1], A[t%2][i][j-1]);
```

- Dependence notations
  - Distance vectors: (1,-1,0), (1,0,0), (1,1,0), (1,0,-1), (1,0,1)
  - Direction vectors
  - Dependence relations as integer sets with affine constraints and existential quantifiers or Presburger formulae — powerful
- Consider the dependence from the write to the third read:
  $A[(t+1)\%2][i][j] \rightarrow A[t'\%2][i'-1][j']$
  Dependence relation: $\{[t,i,j] \rightarrow [t',i',j'] \mid t' = t+1, i' = i+1, j' = j, 0 \le t \le T-1,\ 0 \le i \le N-1, 0 \le j \le N\}$

Preserving Dependences

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                            A[t%2][i][j+1], A[t%2][i][j-1]);
```

- For affine loop nests, these dependences can be analyzed and represented precisely
- **Next step:** Transform while preserving dependences
  - Find execution reorderings that **preserve** dependences and improve performance
  - Execution reordering as a function: $T(\vec{i})$
  - For all dependence relation instances $(\vec{s} \rightarrow \vec{t})$,
    $T(\vec{t}) - T(\vec{s}) \succ \vec{0}$,
    i.e., the source should precede the target even in the transformed space
- What is the structure of **T**?

Valid Transformations

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                           A[t%2][i][j+1], A[t%2][i][j-1]);
```

- Dependences: $(1,0,0)$, $(1,0,1)$, $(1,0,-1)$, $(1,1,0)$, (1,-1,0)
- Validity: $T(\vec{t}) - T(\vec{s}) \succ \vec{0}$, i.e., $T(\vec{t} - \vec{s}) \succ \vec{0}$
- Examples of invalid transformations
  - $T(t,i,j) = (i,j,t)$
  - Similarly, (i,t,j), (j,i,t), $(t+i,i,j)$, $(t+i+j,i,j)$ are all invalid transformations
- Valid transformations
  - (t,j,i), $(t,t+i,t+j)$, $(t,t+i,t+i+j)$
  - However, only some of the infinitely many valid ones are interesting

Generating Loops After Transformation

- Fourier-Motzkin elimination can be used to generate code
  - Successively eliminate old loop variables, and then new loop variables from innermost to outermost, generating bounds for the loop being eliminated at each step.
  - Replace old loop IVs with new ones in the loop body
- More powerful techniques exist to generate more efficient code (fewer/no redundancy in loop bound checks, conditional guards)
- Work out for this example transformation: $(i, j) \rightarrow (i + j, j)$.

Parallelism and Dependence Carrying

- Carrying or satisfying a dependence
- Loop-carried dependence
- A loop is parallel if does not carry any dependences.
- For each dependence, determine the depth at which it is carried
- For constant distance vectors, the depth of the first non-zero dependence component is the depth at which the dependence is satisfied

Synchronization-Free or Communication-free Parallelism

- Number of degrees of synchronization-free parallelim
- m: Dimensionality of the iteration space
- D: Dependence matrix – columns are distance vectors
- m - $rank(D)$ degrees of synchronization-free parallelism
- For any perfect loop nest that has only constant dependences, we can always obtain at least $m - 1$ degrees of parallelism.
- How do you determine or maximize synchronization-free parallelism? Find T (transformation matrix) that satisfies certain properties.
- Find $\vec{t} \neq \vec{0}$ such that $\vec{t}.\vec{d_i} = 0, \forall \vec{d_i}$ (dependence distance vector).

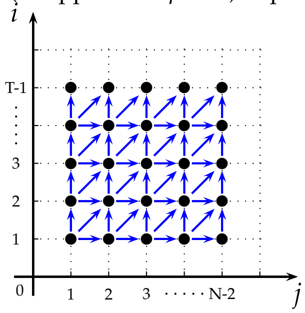
Wavefront Parallelism

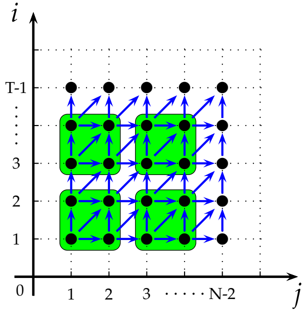
- Synchronization required after execution of a parallel loop
  - A single outer sequential loop with N iterations containing all inner parallel loops will lead to O(N) synchronization
- Refer illustration earlier in this chapter: $(i + j, j)$ mapping for an example
- Connection to *DoAcross* parallelism, as opposed to *DoAll parallelism*?
- It's possible to parallelize using barrier-style synchronization or point-to-point synchronization (between specific pairs of processors)

Outline

Tiling (blocking)

- Partition and execute iteration space in blocks
- A tile is executed atomically
- Benefits: exploits *cache locality* & improves *parallelization* in the presence of synchronization
- **Allows reuse in multiple directions**
- **Reduces frequency of synchronization** for parallelization: synchronization after you execute *tiles* (as opposed to *points*) in parallel





$$(\mathbf{i},\mathbf{j}) \rightarrow (\mathbf{i}/\mathbf{50},\mathbf{j}/\mathbf{50},\mathbf{i},\mathbf{j}); \qquad (\mathbf{i},\mathbf{j}) \rightarrow (\mathbf{i}/\mathbf{50}+\mathbf{j}/\mathbf{50},\mathbf{j}/\mathbf{50},\mathbf{i},\mathbf{j})$$

Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles
  - **Sufficient condition**: All dependence components should be non-negative along dimensions that are being tiled
  - Dependences: (1,0), (1,1), (1,-1)

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    A[(i+1)%2][j] = f(A[i%2][j-1],
            A[i%2][j], A[i%2][j+1]);
```

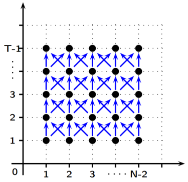


Figure: Iteration space

Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles
  - **Sufficient condition**: All dependence components should be non-negative along dimensions that are being tiled
  - Dependences: (1,0), (1,1), (1,-1)

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    A[(i+1)%2][j] = f(A[i%2][j-1],
                A[i%2][j], A[i%2][j+1]);
```

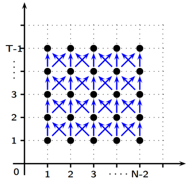


Figure: Iteration space

Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles
  - **Sufficient condition**: All dependence components should be non-negative along dimensions that are being tiled
  - Dependences: (1,0), (1,1), (1,-1)

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    A[(i+1)%2][j] = f(A[i%2][j-1],
                A[i%2][j], A[i%2][j+1]);
```

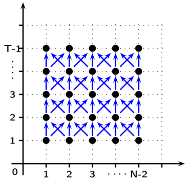
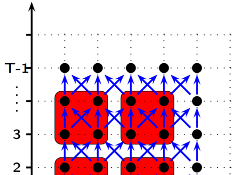


Figure: Iteration space



Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles
  - **Sufficient condition**: All dependence components should be non-negative along dimensions that are being tiled
  - Dependences: (1,0), (1,1), (1,-1)

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    A[(i+1)%2][j] = f(A[i%2][j-1],
                A[i%2][j], A[i%2][j+1]);
```

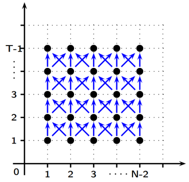
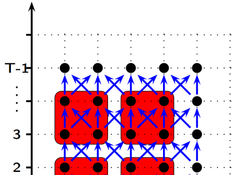
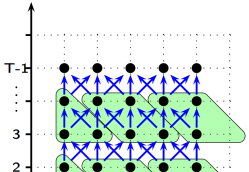


Figure: Iteration space





Tiling (blocking)

- Affine transformations can enable tiling
  - First skew: $T(i, j) = (i, i + j)$

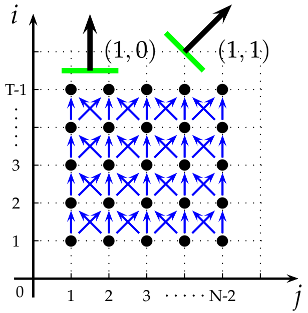


Figure: Original space (i, j)

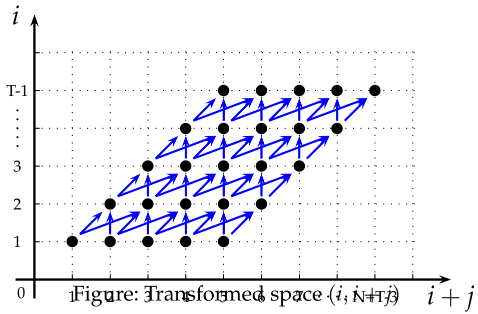


Figure: Transformed space $(i, i + j)$

Tiling (blocking)

- Affine transformations can enable tiling
  - First skew: $T(i,j) = (i, i+j)$
  - Then, apply (rectangular) tiling: $T(i,j) = (i/64, (i+j)/64, i, i+j)$
    - i and $i+j$ are also called *tiling hyperplanes*

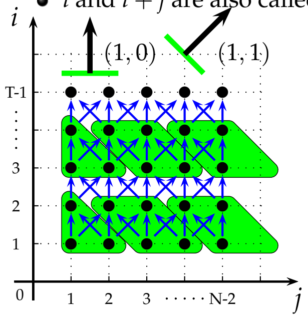


Figure: Original space (i,j)

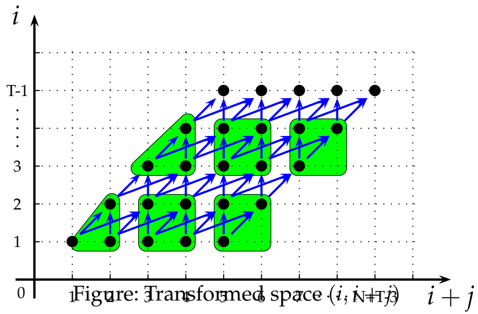


Figure: Transformed space $(i, i+j)$

Back to 3-D example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                           A[t%2][i][j+1], A[t%2][i][j-1]);
```

- What is a good transformation here to improve parallelism and locality?
- Demo
  - Skewing: $(t, t+i, t+j)$
  - Tiling: $(t/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$
  - Tile wavefront: $(t/64+(t+i)/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$

Back to 3-D example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                           A[t%2][i][j+1], A[t%2][i][j-1]);
```

- What is a good transformation here to improve parallelism and locality?
- Demo
  - Skewing: $(t, t+i, t+j)$
  - Tiling: $(t/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$
  - Tile wavefront: $(t/64 + (t+i)/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$

Back to 3-d example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                           A[t%2][i][j+1], A[t%2][i][j-1]);
```

- What is a good transformation here to improve parallelism and locality?
- Demo
  - Skewing: $(t, t+i, t+j)$
  - Tiling: $(t/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$
  - Tile wavefront: $(t/64+(t+i)/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$

Back to 3-D example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                          A[t%2][i][j+1], A[t%2][i][j-1]);
```

- What is a good transformation here to improve parallelism and locality?
- Demo
  - Skewing: $(t, t+i, t+j)$
  - Tiling: $(t/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$
  - Tile wavefront: $(t/64+(t+i)/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$

Other Transformations and Optimizations

- Loop Fusion
- Loop Distribution
- Vectorization
- Explicit copying (Packing)
- Unroll-and-Jam, Register Tiling
- Complementary/enabling transformations for Parallelism
  - Privatization, Scalar expansion, Array Expansion
  - Trade-off between parallelism and memory usage
- Reductions - parallelization and vectorization

Loop Fusion: Validity

- A fine (or finer) grained interleaving of the execution of multiple loop nests
- Validity: fusion is valid if, for every loop being fused, there are no dependences from the first nest body to the second nest body that have a negative component on the loop being fused while not being carried by any outer loops
- Data Dependence Graph (DDG) needed to model "inter-statement" dependences to analyze the above conditions
  - Statements (IR operations or groups of IR operations) are nodes of this graph
  - Each edge corresponds to a dependence from the source node to the target node
  - Directed graph, can have multiple edges between nodes and self edges.
  - Each edge has information on the source and target memory accesses involved in the dependence and additional information.

Fusion: Example

```
// Original code.
// Produces B[i] using another array A.
for (i = 0; i < N - 1; i++)
  B[i] = A[i] + A[i + 1];
// Consumes B[i] to create C[i].
for (i = 0; i < N - 1; i++)
  C[i] = B[i];
```

```
// Fused code.
for (i = 0; i < N - 1; i++) {
  B[i] = A[i] + A[i + 1];
  C[i] = B[i];
}
```

```
// Fusion not valid without shifting the second nest forward by one.
for (i = 0; i < N; i++)
  B[i] = A[i];
// Consumes B[i] to create C[i].
for (i = 0; i < N - 1; i++)
  C[i] = B[i] + B[i + 1];
```

- Fusion can be enabled other transformations: shifting, permutation/interchange
- Fusion can be partial as well, i.e., not fusing all loops
- For partial fusion, consider dependence components up until the loops being fused.

Fusion: Other Examples

```
// Original code.
// Produces B using another array A.
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    B[i][j] = A[i][j];
// Consumes B to create C. Fusion is valid.
// Dependence carried on the fused ‘i‘ loop.
for (i = 0; i < N; i++)
  for (j = 0; j < N - 1; j++)
    C[i][j] = B[i][j] + B[i - 1][j + 1];
```

```
// Original code.
// Produces B using another array A.
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    B[i][j] = A[i][j];
// Consumes B to create C.
for (i = 1; i < N; i++)
  for (j = 0; j < N - 1; j++)
    C[i - 1][j] = B[i][j] + B[i - 1][j];
```

Loop Fusion and Distribution: Costs/Benefits

- Benefits
  1. Improves cache locality: producer-consumer reuse, input reuse
  2. Improves register reuse
  3. Eliminates intermediate arrays and reduces memory consumption
  4. Reduces code size, less control overhead
- Disadvantages
  1. Reduces effective cache capacity available for each of components fused: cache capacity misses
  2. Increases the risk of conflict misses
  3. Can lead to loss of parallelism, loss of tilability, or loss of vectorizability
  4. Increases hardware prefetch stream utilization; can lead to lower prefetching performance

Loop Distribution

- Loop distribution is the inverse of fusion
- Two operations/statements part of the same strongly connected component of the data dependence graph can't be distributed
- Distribution at the inner level or partial distribution: consider only a part of the DDG, discarding dependences carried on outer loops that aren't being considered for distribution.
- Maximal distribution: distribute out all strongly connnected components of a loop nest.
- Disadvantages of fusion are the benefits of distribution

VECTORIZATION

- A fine-grained parallelization: single instruction on multiple data (SIMD)
- Vectorization, SIMDization used synonymously today
- An efficient form of parallelization with minimal additional hardware resources
- Reduction in the number of instructions executed
- The instructions that form a vector can come from a loop body (“superword-level parallelism”) or from a loop (“loop vectorization”)

Loop Vectorization: Examples

```
// Vectorizable loop.
for (i = 0; i < N; i++)
  C[i] = A[i] + B[i];


// Non-vectorizable loop.
for (i = 2; i < N; i++)
  A[i] = A[i - 1] + A[i - 2];


// A loop doesn’t have to be parallel to be vectorizable.
// Loop i is vectorizable despite not being parallel and despite
// carrying a short loop dependence. No dependence cycle.
for (i = 0; i < N; i++) {
  C[i + 1] = A[i] ∗ B[i];
  D[i] = C[i] + X[i];
}
// Vectorizing a loop body like this can also be viewed as tiling by vector
// width, distributing the intra-tile loops, and vectorizing them.
```

Loop Vectorization: Validity

- A loop can be vectorized only if there is no dependence cycle betweeen the instructions that spans less than the “vector width” iterations.
- Contiguity: Data being loaded for a vector may need to be contiguous in memory; depends on hardware
- Alignment: data may have to be aligned depending on the hardware – modern general-purpose processors typically don’t have an alignment requirement
- Performance of aligned vs unaligned memory operations

Vectorization: Example

```
// Original code.
affine.for %i = 0 to 4096 {
 affine.for %j = 0 to 4096 {
  affine.for %k = 0 to 4096 {
   %lhs = affine.load %A[%i, %k] : memref<4096x4096xf32>
   %rhs = affine.load %B[%k, %j] : memref<4096x4096xf32>
   %in = affine.load %C[%i, %j] : memref<4096x4096xf32>
   %product = arith.mulf %lhs, %rhs : f32
   %acc = arith.addf %in, %product : f32
   affine.store %acc, %C[%i, %j] : memref<4096x4096xf32>
  }
 }
}


// Interchanged %j to innermost and vectorized 8-way along the %j loop.
affine.for %i = 0 to 4096 {
 affine.for %k = 0 to 4096 {
  affine.for %j = 0 to 4096 step 8 {
   %lhs = affine.load %A[%i, %k] : memref<4096x4096xf32>
   %v_lhs = vector.splat %lhs : vector<8xf32>
   %v_rhs = affine.vector_load %B[%k, %j] : memref<4096x4096xf32>
   %product = arith.mulf %v_lhs, %v_rhs : vector<8xf32>
   %in = affine.vector_load %C[%i, %j] : memref<4096x4096xf32>
   %acc = arith.addf %in, %product : vector<8xf32>
   affine.vector_store %acc, %C[%i, %j] : memref<4096x4096xf32>
  }
 }
}
```

Explicit Copying or Packing

- Typically performed in conjunction with tiling
- Pack data being accessed by a 'tile' into a contiguous buffer that fits in cache/fast memory
- 'Compute' tile reads from packed input buffers and writes out to a packed buffer; unpack output buffer.
- Benefits
  1. Eliminates conflicts misses and thus improves cache locality
  2. Reduces TLB misses
  3. Improves prefetching performance (fewer hardware prefetch streams used)
- Packing involves overhead (copy-in and copy-out)
- Reference: see packing scheme for high-performance matrix-matrix multiplication in this illustration:
  Analytical Modeling is Enough for High Performance BLIS, Low et al., ACM TOMS 2016.

Unroll-and-jam or Register tiling

- Improves register reuse
- Multi-dimensional unroll-and-jam (multiple loops) can be performed to simultaneously exploit register reuse along multiple dimensions
- Can be thought of as tiling for register locality except that the tiles are small (variables being reused to fit in registers ideally) and the tile is fully unrolled.
- Improves the compute to load/store operation ratio – extremely important for high-performance on modern architectures
- Sufficient: if it is valid to make a loop the innermost loop, it is valid to unroll-and-jam it.
- More precise: unroll-and-jam is valid iff stripminng the loop by the unroll-and-jam factor and bringing the intra-tile loop to the innermost position is valid
- Multi-dimensional unroll-and-jam (multiple loops)

Unroll-and-jam or Register tiling (continued)

- For a matrix-matrix multiplication in the canonical *ijk* form, work out the improvement in compute to load/store ratio when unroll-and-jamming i and j loops with factors U_i and U_j respectively.
- Assume a register budget of 16 registers in one case and 32 registers in another.

Reductions

- Reductions can be parallelized
- Reductions can be vectorized

```
s = 0;
for (i = 0; i < N; i++)
  s += A[i];
```

A composition of transformations

```
for (i = 1 i < N; i++)
  // S1.
  B[i] = A[i];

for (i = 1; i < N; i++)
  // S2.
  C[i - 1] = B[i] + B[i - 1]
```

- Original ordering: $T_{S_1}(i) = (0, i)$, $T_{S_2}(i) = (1, i)$
- Fused + Tiled + Innermost loop distribution
  - Produce a chunk of A and consume it before a new chunk is produced
  - Transformation: $T_{S_1}(i) = (i/32, 0, i)$, $\quad T_{S_2}(i) = (i/32, 1, i)$.

```
for (t1=0;t1<=floord(N-1,32);t1++) {
  for (t3=max(1,32*t1;t3<=min(N-1,32*t1+31);t3++)
    B[t3] = A[t3];
  for (t3=max(1,32*t1);t3<=min(N-1,32*t1+31);t3++)
    C[t3 - 1] = B[t3] + B[t3 - 1];
}
```

  - Provides cache locality while also providing parallelism and vectorization.
  - Either locality or parallelism/vectorizability would have otherwise been lost with only fusion or only parallelizing without any fusion.

ALGORITHMS TO FIND TRANSFORMATIONS

- **The history**
  - A data locality optimizing algorithm, Wolf and Lam, PLDI 1991: Improve locality through unimodular transformations
    - Characterize self-spatial, self-temporal, and group reuse
    - Find unimodular transformations (permutation, reversal, skewing) to transform to permutable loop nests with reuse, and subsequently tile them
- Several advances on polyhedral transformation algorithms through 1990s and 2000s: Feautrier [1991–1992], Lim and Lam (Affine Partitioning) [1997–2001], Pluto [2008–2015]
- **The Present**
  - Polyhedral framework provides a powerful mathematical abstraction (away from the syntax)
  - A number of new techniques, open-source libraries and tools have been developed and are actively maintained
  - Affine abstractions and infrastructure in MLIR

OUTLINE

MLIR

- Open-sourced by Google in Apr 2019
- ML in MLIR: Multi-level
- Ability to represent code at multiple levels in a unified way
- First class abstractions for multi-dimensional arrays (tensors), loop nests, affine maps/sets, and more

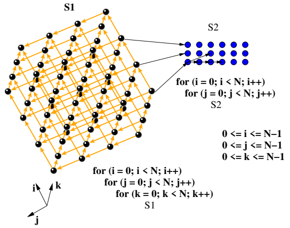
Outline

MLIR: Multi-level Intermediate Representation

1. Ops on tensor types 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



```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    S2
```

```
0 <= i <= N−1
0 <= j <= N−1
0 <= k <= N−1
```

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      S1
```

```
#map0 = affine_map<(d0) −> (d0)>
#map1 = affine_map<(d0) −> (d0 + 4)>
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 %lhs[%ii, %kk] : memref<8 x 8 x f32>
            %6 = affine.load %rhs[%kk, %jj] : memref<8 x 8 x f32>
            %7 = affine.load %out[%ii, %jj] : memref<8 x 8 x f32>
            %8 = arith.mulf %5, %6 : f32
            %9 = arith.addf %7, %8 : f32
            affine.store %9, %out[%ii, %jj] : memref<8 x 8 x f32>
          }
        }
      }
    }
  }
}
```

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

MLIR Design Principles

1. Round-trippable textual format
2. Ability to represent code at multiple levels
3. Unified representation for all the levels

MLIR Design Principles

1. Round-trippable textual format
2. Ability to represent code at multiple levels
3. Unified representation for all the levels

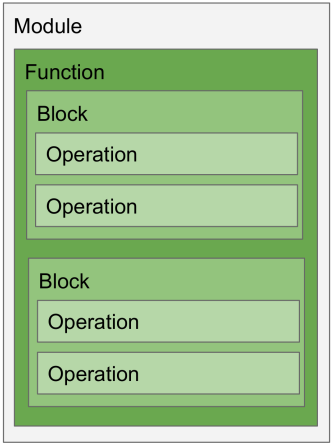
MLIR Design Principles

1. Round-trippable textual format
2. Ability to represent code at multiple levels
3. Unified representation for all the levels

MLIR - Basic Concepts

- SSA, typed, three address
- Module/Function/Block/Operation structure
- Operations can hold a “region”, which is a list of blocks

```
func.func @test(%arg: i32) {
  %x = call @thing_to_call(%arg) : (i32) -> i32
  cf.br ^bb1
^bb1:
  %y = arith.addi %x, %x : i32
  return %y : i32
}
```



SSA REPRESENTATION

- Functional SSA representation
- No ϕ nodes
- Instead, blocks take arguments

```
func.func @condbr_simple() -> (i32) {
  %cond = "foo"() : () -> i1
  %a = "bar"() : () -> i32
  %b = "bar"() : () -> i64
  cf.cond_br %cond, ^bb1(%a : i32), ^bb2(%b : i64)

^bb1(%x : i32):
  %w = "foo_bar"(%x) : (i32) -> i64
  cf.br ^bb2(%w: i64)

^bb2(%y : i64):
  %z = "abc"(%y) : (i64) -> i32
  return %z : i32
}
```

MLIR Operations

- Operations always have a name and source location info
- Operations may have:
  - Arbitrary number of SSA results and operands
  - Attributes: guaranteed constant values
  - Block operands: e.g. for branch operations
  - Regions: discussed later
  - Custom printing/parsing - or use the more verbose generic syntax

```
%size = tensor.dim %T, 1 : tensor<1024x? x f32>
// Dimension to extract is a guaranteed integer constant, an attribute .
%x = memref.alloc() : memref<1024x64xf32>
%y = affine.load %x[%a, %b] : memref<1024x64xf32>
```

Operations with Regions

- An MLIR Region is a list of blocks
- Operations in MLIR can have nested regions

```
%2 = xla.fusion (%0 : tensor<f32>,
                 %1 : tensor<f32>) : tensor<f32> {
^bb0(%a0 : tensor<f32>, %a1 : tensor<f32>):
  %x0 = xla.add %a0, %a1 : tensor<f32>
  %x1 = xla.relu  %x0 : tensor<f32>
  return %x1
}
```

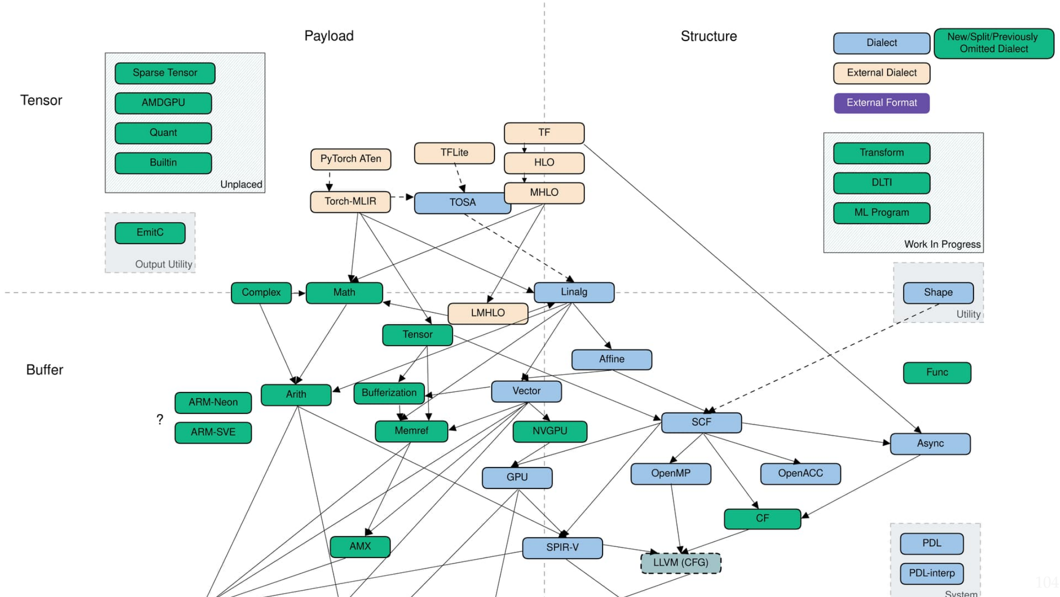
```
func.func @loop_nest_unroll(%arg0: index) {
  affine.for %arg1 = 0 to 100 step 2 {
    affine.for %arg2 = 0 to #map1(%arg0) {
      %0 = "foo"()  :  ()  -> i32
    }
  }
  return
}
```

- Can be used to represent:
  - functional control flow
  - fusion nodes
  - closures/lambdas
  - structured looping/conditional constructs (*for*, *if*, *while*)
  - Parallelism abstractions like OpenMP
  - Launch/dispatch kernel abstractions `gpu.launch`

Dialects in MLIR

- A collection of operations and types suitable for a specific task
- Typically correspond to a programming model, frontend, or a backend
- Example dialects: TensorFlow dialect, LLVM dialect, Affine dialect, NVIDIA GPU dialect
- You can have a mix of dialects

Current Dialects in MLIR



Outline

Affine expressions in MLIR

- Affine for functions is linear + constant
  - Addition of identifiers, multiplication with a constant, floordiv, mod, ceildiv with respect to a positive constant
- Examples of affine functions of i, j:
  $i + j$, $2i - j$, $i + 1$, $2i + 5$,
  $i/128 + 1$, $i\%8$, $(i + j)/8$,
  $((d0 * 9216 + d1 * 128) \text{ mod } 294912) \text{ floordiv } 147456$
- Not affine: ij, i/j, j/N, $i^2 + j^2$, $a[j]$

Affine Maps

- An affine map maps zero or more identifiers to one or more result affine expressions

$$
\begin{aligned}
\#map1 &= (d0) \rightarrow ((d0 \text{ floordiv } 4) \bmod 2) \\
\#map2 &= (d0) \rightarrow (d0 - 4) \\
\#map3 &= (d0) \rightarrow (d0 + 4) \\
\#map4 &= (d0, d1) \rightarrow (d0 * 16 - d1 + 15) \\
\#map5 &= (d0, d1, d2, d3) \rightarrow (d2 - d0 * 16, d3 - d1 * 16)
\end{aligned}
$$

- Why affine maps? What can they express?
  - Loop IV mappings for nearly every useful loop transformation, data layout transformations, placement functions / processor mappings / distributions: block, cyclic, block-cyclic, multi-dimensional array subscripts, loop bound expressions, conditionals

Where are affine maps Used in MLIR?

1. IV remappings: to map old IVs to new IVs

(i, j)	Identity
(j, i)	Interchange
$(i, i + j)$	Skew j
$(2i, j)$	Scale i by two
$(i, j + 1)$	Shift j
$(\lfloor \frac{i}{32} \rfloor, \lfloor \frac{j}{32} \rfloor, i, j)$	Tile (rectangular)
$\ldots$	

2. Loop bounds
3. Memref access subscripts
4. As an attribute for any operation

```
#map = (d0) −> (2∗d0 − 1)

affine.for %i = 0 to #map(%N) {
  affine.for %j = 0 to 3 {
    %v = affine.load %0[%i + %j] : memref<100xf32>
    "op1"(%v) : (f32) −> ()
  }
}
%w = "op"(%s, %t) {map: affine_map<(d0, d1) −> (d1, d0)>}
```

Polyhedral Structures in the IR

1. Affine expressions
   - Eg: $(d0 + 1) \bmod 2$
2. Affine maps
   - Eg: $(d0, d1) \rightarrow (d1, d0/128, d0 \bmod 128)$
3. Integer sets
   - Eg: $\{(d0, d1)[s1] : d0 \geq 0, d0 \leq s1, d1 == 512\}$
4. Affine apply operation (`affine.apply`)

```
%a = affine.apply (d0, d1) -> (d0 + d1) (%i, %j)
```

5. Affine ‘for‘ operation (`affine.for`)
6. Affine ‘if‘ operation (`affine.if`)

Polyhedral Structures in the IR

1. Affine expressions
   - Eg: $(d0 + 1) \bmod 2$
2. Affine maps
   - Eg: $(d0, d1) \rightarrow (d1, d0/128, d0 \bmod 128)$
3. Integer sets
   - Eg: $\{(d0, d1)[s1] : d0 \geq 0, d0 \leq s1, d1 == 512\}$
4. Affine apply operation (`affine.apply`)

```
%a = affine.apply (d0, d1) -> (d0 + d1) (%i, %j)
```

5. Affine ‘for‘ operation (`affine.for`)
6. Affine ‘if‘ operation (`affine.if`)

Types Relevant for Dense Matrices/Tensors

1. *tensor* A value that is a multi-dimensional array of elemental values

```
%d = "tf.Add"(%e, %f) : (tensor<?x42x?xf32>, tensor<?x42x?xf32>) -> tensor<?x42x?xf32>
```

2. *memref* A buffer in memory or a view on a buffer, has a layout map, memory space qualifier, symbols bound to its dynamic dimensions

```
%N = affine.apply (d0) -> (8 * (d0 ceildiv 8)) (%S)
%M = affine.apply (d0) -> (2 * d0) (%N)
#tmap = affine_map<(d0, d1) -> (d1 floordiv 32, d0 floordiv 128, d1 mod 32, d0 mod 128)>
#shift = affine_map<(d0, d1)[s0, s1] -> (d0 + s0, d1 + s1)>
%A = memref.alloc() : memref<1024x64xf32, #tmap, 0>
%B = memref.alloc(%M, %N)[%x, %y] : memref<?x?xf32, #tmap, 1>
%C = memref.alloc(%M, %M)[%x, %y] : memref<?x?xf32, #shift, 1>
```

INTEGER SETS

- An integer set is primarily used for conditionals
- It is also powerful as an attribute to specify constraints on symbols (esp. shape symbols)

```
// An example two-dimensional integer set with two symbols.
#set = affine_set<(d0, d1)[s0, s1]
   : d0 >= 0, -d0 + s0 - 1 >= 0, d1 >= 0, -d1 + s1 - 1 >= 0>

affine.if #set(%i, %j)[%M, %N] {
  %v = affine.load %A[%i] : memref<256xf32>
}
```

Outline

Analyses and Transformations: what's curently present

- Several techniques are available:
  - GCD test
    $2i + 4j - 8k - 1 = 0 \quad \rightarrow$ No solution
  - GCD tightening:
    $16i \geq 16j - 15, \; 16i \leq 16j \quad \rightarrow$ i = j
  - Gaussian elimination
    $i = j - 1, \; 0 \leq j \leq i$
  - Fourier-Motzkin elimination: eliminate a variable from a system of linear inequalities
    $i \leq j + 1, \; j = k, \; k \leq 16, \; i \geq 32$
- FlatAffineConstraints
- Fast Presburger library (FPL)

Generalized Slicing-based Loop Fusion

- A generalized slicing-based loop fusion approach
- Can trade off redundant computation for locality / memory minimization
- Post fusion, forwarding of ‘affine.store‘ to affine.load, elimination of intermediate arrays can be performed
- Fixed size local buffers are created when possible to pass intermediate data

```
affine.for %i = 0 to 64 {
  %v = affine.load %in[%i] : memref<64xf32>
  affine.store %v, %out[%i floordiv 4, %i mod 4]
      : memref<16x4xf32>
}

affine.for %i = 0 to 16 {
  affine.for %j = 0 to 4 {
    %w = affine.load %out[%i, %j] : memref<16x4xf32>
    "foo"(%w) : (f32) -> ()
  }
}
```

```
affine.for %i = 0 to 16 {
  affine.for %j = 0 to 4 {
    %v = affine.load %in[4 * %i + %j] : memref<64xf32>
    "foo"(%v) : (f32) -> ()
  }
}
```

Outline

Compiler Backend

1. Instruction Selection
2. Instruction Scheduling
3. Register Allocation

Register Allocation

- Machines have a limited number of registers
- Register Allocation: Determine which variables should be allocated registers and assign them registers
- Objectives
  - Minimize "spilling"
  - Efficiency: time complexity, performance in practice

REGISTER ALLOCATION: DEFINITIONS

- A variable is **live** from its definition to its last use
- Two variables cannot be allocated the same register if they are both simulataneously live
- Such simultaneously live variables are said to **interfere**
- **Spilling** saves a values from a register to memory; a register is freed
- A variables that has not been updated can be spilled without a store to memory

Live Ranges and Interference Graph

- Perform analysis to compute liveness information
- From live ranges, construct an **interference graph**
- Variables are nodes of the graph (each node has a live range)
- An edge between nodes iff the associated variables' live ranges interfere
- Color the interference graph such that no two adjacent vertices have the same color
- **k** registers: find a **k-colouring** for the interference graph
- Registers are colours
- NP-complete problem in general

Chaitin's algorithm: Register Allocation by Graph Coloring

1. While $\exists$ vertices with $< k$ neighbours in G:
   - Pick any vertex n such that $deg(n) < k$ and put it on the stack
   - Remove n and all edges incident to it from G
2. If G is non-empty with $deg(v) \geq k, \forall v \in G$ then:
   Pick vertex v (using a heuristic), spill live range of v
   Remove vertex v and edges from G, put v on the "spill list"
   Go to step 1
3. If the spill list is not empty, insert spill code, then rebuild the interference graph and try to allocate, again
4. Otherwise, successively pop vertices off the stack and colour them in the lowest colour not used by some neighbour.

Linear Scan Algorithm

- Linearization of basic blocks
- Approximate register allocation as the coloring of **interval graphs**
- **Live interval** A sequence of instructions, outside of which a variable v is never live.
- The algorithm
  1. Walk intervals in the sorted increasing order of start points
  2. Maintain a pool of available registers, determine expired intervals, free registers, and allocate from the pool
  3. When no registers are available, spill the interval that has the latest finish point (other heuristics possible)
- Complexity: $O(VlogR)$: V variables, R registers

Recent Work

- Copy Coalescing
- Register allocation under SSA: interference graphs are chordal graphs. Chordal graphs can be coloured efficiently
- Register allocation has to be solved in combination with instruction scheduling and code generation
- Classic phase ordering problem between register allocation and instruction scheduling