Challenges in Decompilation and Reverse Engineering of CUDA-based Kernels

Nicolò Altamura

@nicolodev
nicolo.dev
seekbytes@protonmail.com

About me

- security engineer at emproof
- code deobfuscation by night
- recently graduated at University of Verona

Overview

Introduction to CUDA

Microarchitectural details

Decompilation

Why CUDA reverse engineering matters

- AI-oriented applications include code for CUDA architecture

Why CUDA reverse engineering matters

- AI-oriented applications include code for CUDA architecture
- new parallel paradigm poses new challenges

Why CUDA reverse engineering matters

- AI-oriented applications include code for CUDA architecture
- new parallel paradigm poses new challenges
- opaque semantics

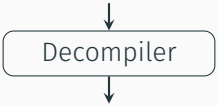
Why CUDA reverse engineering matters

- AI-oriented applications include code for CUDA architecture
- new parallel paradigm poses new challenges
- opaque semantics

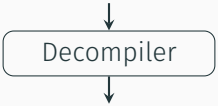
Missed opportunity in binary research

```
.text._Z6matMulPKfS0_Pfi (PROGBITS) section started  {0x600-0xc80}
00000600  02 7a 01 00 00 0a 00 00-00 0f 00 00 00 c4 0f 00-89 f3 ff ff ff 00 00 00-ff 00 0e 00 00 e2 0f 00  .z..............................
00000620  19 79 09 00 00 00 00 00-00 25 00 00 00 22 0e 00-02 7a 06 00 00 5e 00 00-00 0f 00 00 00 e4 0f 00  .y.......%..."...z...^..........
00000640  02 72 15 00 ff 00 00 00-00 0f 00 00 00 e2 0f 00-19 79 0c 00 00 00 00 00-00 21 00 00 00 22 0e 00  .r.............y.......!..."..
00000660  0c 78 00 06 01 00 00 00-70 62 f2 03 00 c6 0f 00-19 79 03 00 00 00 00 00-00 26 00 00 00 68 0e 00  .x......pb.......y.......&...h..
00000680  19 79 02 00 00 00 00 00-00 22 00 00 00 62 0e 00-24 78 00 09 10 00 00 00-0c 02 8e 07 00 ca 1f 00  .y.......".....b..$x............
000006a0  0c 7a 00 00 00 5e 00 00-70 62 f0 03 00 e2 0f 00-24 78 03 03 10 00 00 00-02 02 8e 07 00 ca 2f 00  .z...^..pb......$x............/.
000006c0  0c 7a 00 03 00 5e 00 00-70 66 70 00 00 e2 0f 00-47 99 00 00 e0 04 00 00-00 00 80 03 00 f6 0f 00  .z...^..pfp.....G...............
```

```
.text._Z6matMulPKfS0_Pfi (PROGBITS) section started  {0x600-0xc80}
00000600  02 7a 01 00 00 0a 00 00-00 0f 00 00 00 c4 0f 00-89 f3 ff ff ff 00 00 00-ff 00 0e 00 00 e2 0f 00  .z..............................
00000620  19 79 09 00 00 00 00 00-00 25 00 00 00 22 0e 00-02 7a 06 00 00 5e 00 00-00 0f 00 00 00 e4 0f 00  .y.......%..."...z...^..........
00000640  02 72 15 00 ff 00 00 00-00 0f 00 00 00 e2 0f 00-19 79 0c 00 00 00 00 00-00 21 00 00 00 22 0e 00  .r.............y.......!..."..
00000660  0c 78 00 06 01 00 00 00-70 62 f2 03 00 c6 0f 00-19 79 03 00 00 00 00 00-00 26 00 00 00 68 0e 00  .x......pb.......y.......&...h..
00000680  19 79 02 00 00 00 00 00-00 22 00 00 00 62 0e 00-24 78 00 09 10 00 00 00-0c 02 8e 07 00 ca 1f 00  .y.......".....b..$x............
000006a0  0c 7a 00 00 00 5e 00 00-70 62 f0 03 00 e2 0f 00-24 78 03 03 10 00 00 00-02 02 8e 07 00 ca 2f 00  .z...^..pb......$x............/.
000006c0  0c 7a 00 03 00 5e 00 00-70 66 70 00 00 00 e2 0f 00-47 99 00 00 e0 04 00 00-00 00 80 03 00 f6 0f 00  .z...^..pfp.....G...............
```



```
.text._Z6matMulPKfS0_Pfi (PROGBITS) section started  {0x600-0xc80}
00000600  02 7a 01 00 00 0a 00 00-00 0f 00 00 00 c4 0f 00-89 f3 ff ff ff 00 00 00-ff 00 0e 00 00 e2 0f 00  .z..............................
00000620  19 79 09 00 00 00 00 00-00 25 00 00 00 22 0e 00-02 7a 06 00 00 5e 00 00-00 0f 00 00 00 e4 0f 00  .y.......%..."...z...^..........
00000640  02 72 15 00 ff 00 00 00-00 0f 00 00 00 e2 0f 00-19 79 0c 00 00 00 00 00-00 21 00 00 00 22 0e 00  .r.............y.......!..."..
00000660  0c 78 00 06 01 00 00 00-70 62 f2 03 00 c6 0f 00-19 79 03 00 00 00 00 00-00 26 00 00 00 68 0e 00  .x......pb.......y.......&...h..
00000680  19 79 02 00 00 00 00 00-00 22 00 00 00 62 0e 00-24 78 00 09 10 00 00 00-0c 02 8e 07 00 ca 1f 00  .y.......".....b..$x............
000006a0  0c 7a 00 00 00 5e 00 00-70 62 f0 03 00 e2 0f 00-24 78 03 03 10 00 00 00-02 02 8e 07 00 ca 2f 00  .z...^..pb.....$x............/.
000006c0  0c 7a 00 03 00 5e 00 00-70 66 70 00 00 e2 0f 00-47 99 00 00 e0 04 00 00-00 00 80 03 00 f6 0f 00  .z...^..pfp.....G..............
```




```
int64_t _Z6matMulPKfS0_Pfi(int32_t arg1 @ r14, int32_t arg2 @ r33, int32_t arg3 @ r44, bool arg4 @ p5, bool arg5 @ p9)

    void* param_0 = _Z6matMulPKfS0_Pfi_param_0
    void* param_1 = _Z6matMulPKfS0_Pfi_param_1
    void* param_2 = _Z6matMulPKfS0_Pfi_param_2
    void* param_3 = _Z6matMulPKfS0_Pfi_param_3
    int32_t param_5 = _Z6matMulPKfS0_Pfi_param_5
    int32_t param_6 = _Z6matMulPKfS0_Pfi_param_6
    int32_t tid.x
    int32_t ctaid.x
    int32_t ntid.x

    if (param_5 s>= 1)
        int32_t r31_1 = ntid.x * ctaid.x + tid.x
        int32_t r1_1 = r31_1 * param_5
        int32_t r2_1 = r31_1 * param_6

        if (param_6 s>= 0)
            int32_t r15 = param_6 & 3
            int32_t r50 = 0
```

Introduction to CUDA

matrix_sum

Let's start with a simple problem: **addition of matrices**.

```
int main(){
    int A[M][N], B[M][N], C[M][N];
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

matrix_sum

Let's start with a simple problem: **addition of matrices**.

```
int main(){
    int A[M][N], B[M][N], C[M][N];
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

Execution on CPU:

matrix_sum

Let's start with a simple problem: **addition of matrices**.

```
int main(){
    int A[M][N], B[M][N], C[M][N];
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

Execution on CPU:

1. `C[0][0] = A[0][0] + B[0][0];`

matrix_sum

Let's start with a simple problem: **addition of matrices**.

```
int main(){
    int A[M][N], B[M][N], C[M][N];
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

Execution on CPU:

1. `C[0][0] = A[0][0] + B[0][0];`
2. `C[0][1] = A[0][1] + B[0][1];`

matrix_sum

Let's start with a simple problem: **addition of matrices**.

```
int main(){
    int A[M][N], B[M][N], C[M][N];
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

Execution on CPU:

1. `C[0][0] = A[0][0] + B[0][0];`
2. `C[0][1] = A[0][1] + B[0][1];`
3. `C[0][2] = A[0][2] + B[0][2];`

matrix_sum

Let's start with a simple problem: **addition of matrices**.

```
int main(){
    int A[M][N], B[M][N], C[M][N];
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

Execution on CPU:

1. `C[0][0] = A[0][0] + B[0][0];`
2. `C[0][1] = A[0][1] + B[0][1];`
3. `C[0][2] = A[0][2] + B[0][2];`
4. ...

matrix_sum

```
int main(){
    int A[M][N], B[M][N], C[M][N];
    // Compute the sum of the two
     ↪  matrices
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

matrix_sum

```c
int main(){
    int A[M][N], B[M][N], C[M][N];
    // Compute the sum of the two
     ↪  matrices
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

- Bottleneck for sequential execution

matrix_sum

```c
int main(){
    int A[M][N], B[M][N], C[M][N];
    // Compute the sum of the two
     ↪  matrices
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

- Bottleneck for sequential execution
- Execution time:

$$O(M \cdot N)$$

matrix_sum

```c
int main(){
    int A[M][N], B[M][N], C[M][N];
    // Compute the sum of the two
     ↪  matrices
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

- Bottleneck for sequential execution
- Execution time:

$$O(M \cdot N)$$

- Every calc is independent

matrix_sum

```
int main(){
    int A[M][N], B[M][N], C[M][N];
    // Compute the sum of the two
     ↪  matrices
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
```

- Bottleneck for sequential execution
- Execution time:

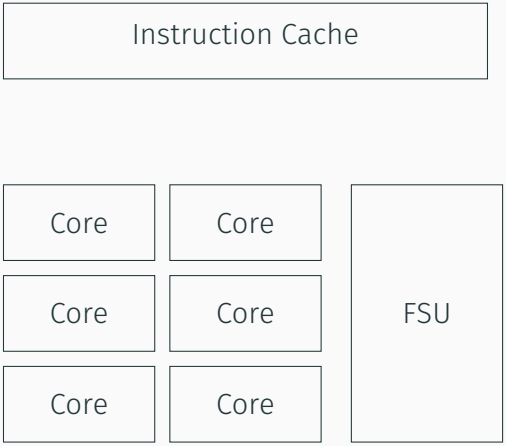
$$O(M \cdot N)$$

- Every calc is independent

Idea: execute every operation *in parallel*

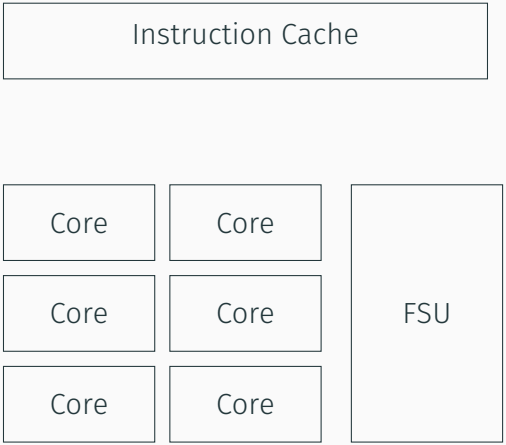
Ideally: 1 computation ≈ 1 ALU

Physical view: streaming multiprocessors



From idea to hardware implementation:

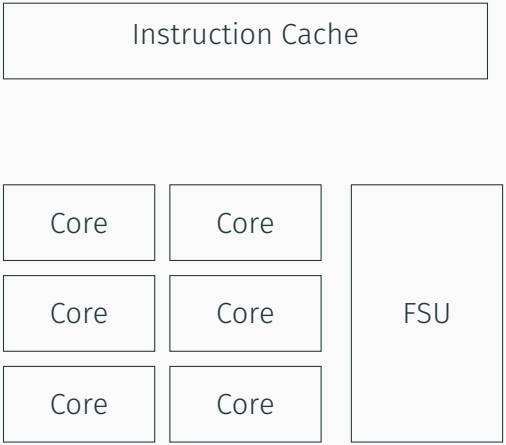
Physical view: streaming multiprocessors



From idea to hardware implementation:

- array of Streaming Multiprocessors (SM)

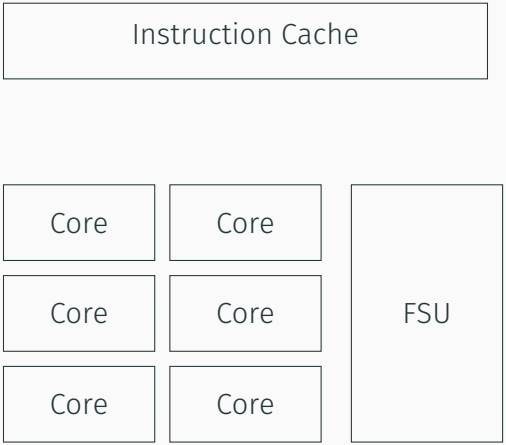
Physical view: streaming multiprocessors



From idea to hardware implementation:

- array of Streaming Multiprocessors (SM)
- every SM has multiple cores

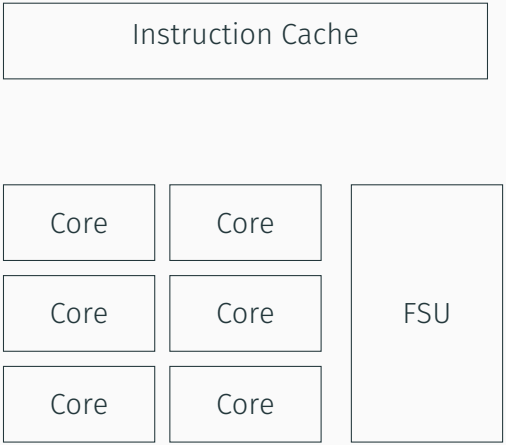
Physical view: streaming multiprocessors



From idea to hardware implementation:

- array of Streaming Multiprocessors (SM)
- every SM has multiple cores
- each core is able to execute operation in parallel via threads

Physical view: streaming multiprocessors

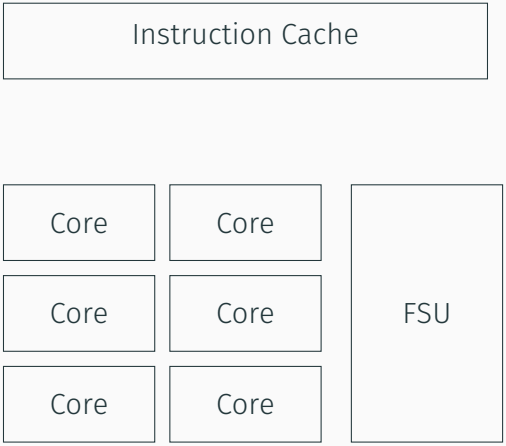


From idea to hardware implementation:

- array of Streaming Multiprocessors (SM)
- every SM has multiple cores
- each core is able to execute operation in parallel via threads

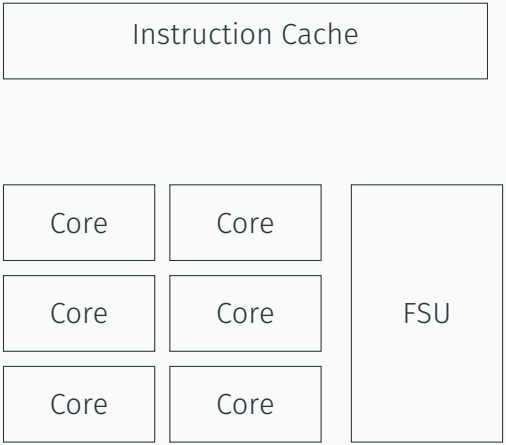
Welcome to CUDA

Physical view: streaming multiprocessors/2



CUDA includes:

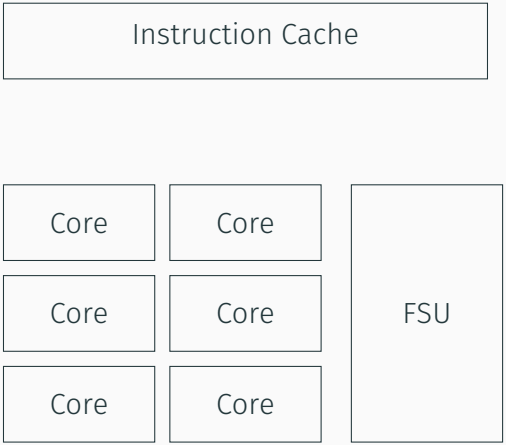
Physical view: streaming multiprocessors/2



CUDA includes:

- synchronization primitives

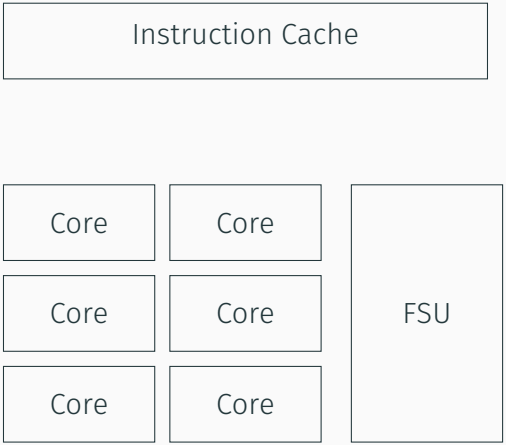
Physical view: streaming multiprocessors/2



CUDA includes:

- synchronization primitives
- scheduler

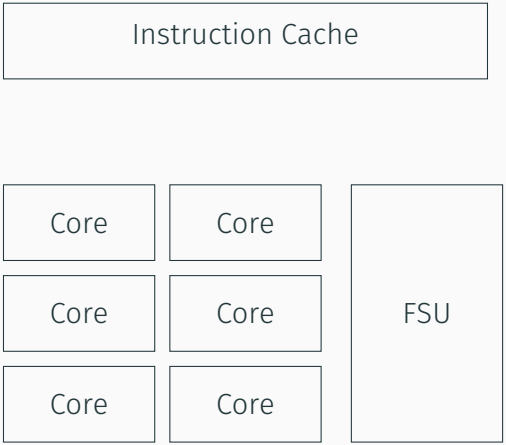
Physical view: streaming multiprocessors/2



CUDA includes:

- synchronization primitives
- scheduler
- specialized ALUs

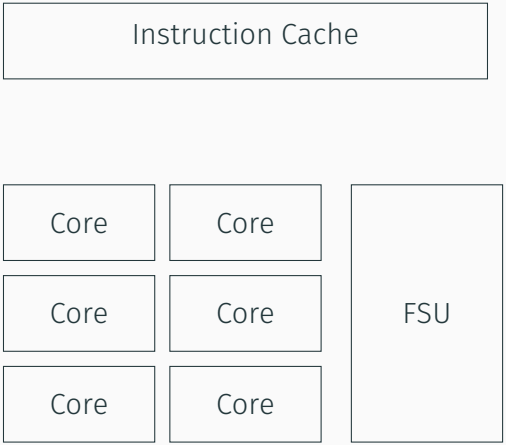
Physical view: streaming multiprocessors/2



CUDA includes:

- synchronization primitives
- scheduler
- specialized ALUs
- hardware caches (instructions/constants)

Physical view: streaming multiprocessors/2



CUDA includes:

- synchronization primitives
- scheduler
- specialized ALUs
- hardware caches (instructions/constants)

How do we exploit this?

Host-device paradigm

- host (CPU) ↔ device (GPU) paradigm

```
__global__ void matrix_sum(int *A, int *B, int
↪  *C, int N, int M){
    // Compute the sum of the two matrices
    int row = blockIdx.y * blockDim.y +
     ↪  threadIdx.y;
    int col = blockIdx.x * blockDim.x +
     ↪  threadIdx.x;

    if (row < M && col < N) {
        int idx = row * N + col;
        C[idx] = A[idx] + B[idx];
    }
}
```

Host-device paradigm

- host (CPU) ↔ device (GPU) paradigm
- kernel: routine to be executed in the CUDA device

```
__global__ void matrix_sum(int *A, int *B, int
↪  *C, int N, int M){
    // Compute the sum of the two matrices
    int row = blockIdx.y * blockDim.y +
     ↪  threadIdx.y;
    int col = blockIdx.x * blockDim.x +
     ↪  threadIdx.x;

    if (row < M && col < N) {
        int idx = row * N + col;
        C[idx] = A[idx] + B[idx];
    }
}
```

Host-device paradigm

- host (CPU) ↔ device (GPU) paradigm
- kernel: routine to be executed in the CUDA device
- goal: move the heaviest calculation on the dedicated GPU

```
__global__ void matrix_sum(int *A, int *B, int
↪  *C, int N, int M){
    // Compute the sum of the two matrices
    int row = blockIdx.y * blockDim.y +
    ↪  threadIdx.y;
    int col = blockIdx.x * blockDim.x +
    ↪  threadIdx.x;

    if (row < M && col < N) {
        int idx = row * N + col;
        C[idx] = A[idx] + B[idx];
    }
}
```

Host-device paradigm

- host (CPU) ↔ device (GPU) paradigm
- kernel: routine to be executed in the CUDA device
- goal: move the heaviest calculation on the dedicated GPU
  - host sends data to device

```
__global__ void matrix_sum(int *A, int *B, int
↪  *C, int N, int M){
    // Compute the sum of the two matrices
    int row = blockIdx.y * blockDim.y +
    ↪  threadIdx.y;
    int col = blockIdx.x * blockDim.x +
    ↪  threadIdx.x;

    if (row < M && col < N) {
        int idx = row * N + col;
        C[idx] = A[idx] + B[idx];
    }
}
```

Host-device paradigm

- host (CPU) ↔ device (GPU) paradigm
- kernel: routine to be executed in the CUDA device
- goal: move the heaviest calculation on the dedicated GPU
  - host sends data to device
  - GPU executes the kernel

```
__global__ void matrix_sum(int *A, int *B, int
↪  *C, int N, int M){
    // Compute the sum of the two matrices
    int row = blockIdx.y * blockDim.y +
     ↪  threadIdx.y;
    int col = blockIdx.x * blockDim.x +
     ↪  threadIdx.x;

    if (row < M && col < N) {
        int idx = row * N + col;
        C[idx] = A[idx] + B[idx];
    }
}
```

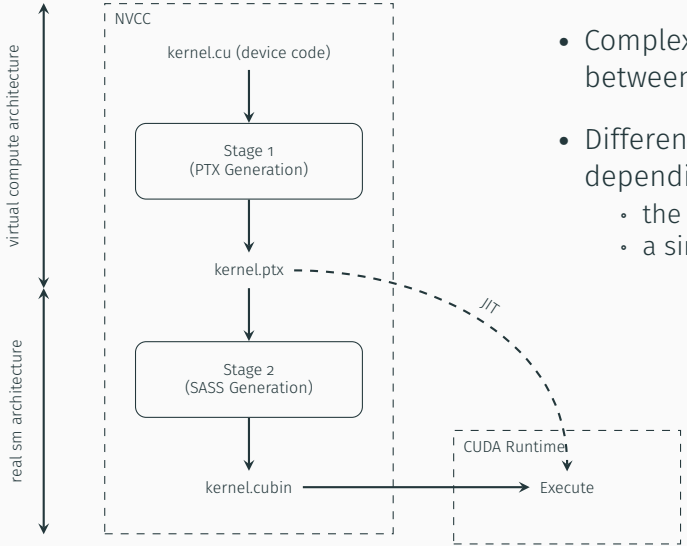
Host-device paradigm

- host (CPU) ↔ device (GPU) paradigm
- kernel: routine to be executed in the CUDA device
- goal: move the heaviest calculation on the dedicated GPU
  - host sends data to device
  - GPU executes the kernel
  - host receives solution

```
__global__ void matrix_sum(int *A, int *B, int
↪  *C, int N, int M){
    // Compute the sum of the two matrices
    int row = blockIdx.y * blockDim.y +
     ↪  threadIdx.y;
    int col = blockIdx.x * blockDim.x +
     ↪  threadIdx.x;

    if (row < M && col < N) {
        int idx = row * N + col;
        C[idx] = A[idx] + B[idx];
    }
}
```

CUDA compilation



- Complex toolchain due to dualism between host and device code
- Different “encoding” of instructions depending on the target:
  - the entire ecosystem: PTX
  - a single GPU: SASS

CUDA languages

PTX: Parallel Thread eXecution

SASS: Streaming ASSembler

CUDA languages

PTX: Parallel Thread eXecution

- text-based low level language

SASS: Streaming ASSembler

CUDA languages

PTX: Parallel Thread eXecution

- text-based low level language
- targets all CUDA architectures

SASS: Streaming ASSembler

CUDA languages

PTX: Parallel Thread eXecution

- text-based low level language
- targets all CUDA architectures
- get translated just-in-time before execution

SASS: Streaming ASSembler

CUDA languages

PTX: Parallel Thread eXecution

- text-based low level language
- targets all CUDA architectures
- get translated just-in-time before execution
- forward compatible with future GPUs

SASS: Streaming ASSembler

CUDA languages

PTX: Parallel Thread eXecution

- text-based low level language
- targets all CUDA architectures
- get translated just-in-time before execution
- forward compatible with future GPUs
- hides micro architecture details

SASS: Streaming ASSembler

CUDA languages

PTX: Parallel Thread eXecution

- text-based low level language
- targets all CUDA architectures
- get translated just-in-time before execution
- forward compatible with future GPUs
- hides micro architecture details

SASS: Streaming ASSembler

- targets a specific CUDA device

CUDA languages

PTX: Parallel Thread eXecution

- text-based low level language
- targets all CUDA architectures
- get translated just-in-time before execution
- forward compatible with future GPUs
- hides micro architecture details

SASS: Streaming ASSembler

- targets a specific CUDA device
- little-to-none details to public

CUDA languages

PTX: Parallel Thread eXecution

- text-based low level language
- targets all CUDA architectures
- get translated just-in-time before execution
- forward compatible with future GPUs
- hides micro architecture details

SASS: Streaming ASSembler

- targets a specific CUDA device
- little-to-none details to public
- include low-level information such as pipeline

CUDA languages

PTX: Parallel Thread eXecution

- text-based low level language
- targets all CUDA architectures
- get translated just-in-time before execution
- forward compatible with future GPUs
- hides micro architecture details

SASS: Streaming ASSembler

- targets a specific CUDA device
- little-to-none details to public
- include low-level information such as pipeline
- actual machine code executed on the GPU

CUDA languages

PTX: Parallel Thread eXecution

- text-based low level language
- targets all CUDA architectures
- get translated just-in-time before execution
- forward compatible with future GPUs
- hides micro architecture details

SASS: Streaming ASSembler

- targets a specific CUDA device
- little-to-none details to public
- include low-level information such as pipeline
- actual machine code executed on the GPU

Today, we focus on PTX 8.0 and Volta series

Exploring the compilation artifacts

Let's compile the example `matrix_sum`:

```
nvcc matrix_sum.cu -o matrix_sum
```

Exploring the compilation artifacts

Let's compile the example `matrix_sum`:

```
nvcc matrix_sum.cu -o matrix_sum
```

In the ELF file generated, we find:

Exploring the compilation artifacts

Let's compile the example `matrix_sum`:

```
nvcc matrix_sum.cu -o matrix_sum
```

In the ELF file generated, we find:

- usual sections of any binary for the host architecture (x86_64)

Exploring the compilation artifacts

Let's compile the example `matrix_sum`:

```
nvcc matrix_sum.cu -o matrix_sum
```

In the ELF file generated, we find:

- usual sections of any binary for the host architecture (x86_64)
- CUDA-specifics sections:

Exploring the compilation artifacts

Let's compile the example `matrix_sum`:

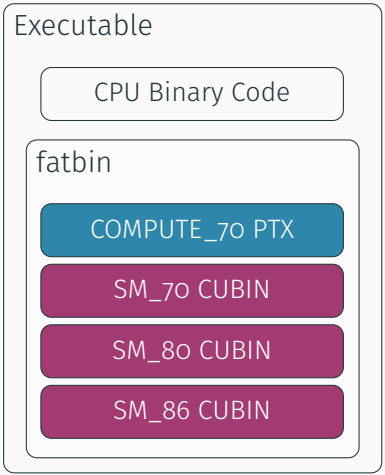
```
nvcc matrix_sum.cu -o matrix_sum
```

In the ELF file generated, we find:

- usual sections of any binary for the host architecture (x86_64)
- CUDA-specifics sections:
  - `.nv_fatbin`: the FatBinary
  - `.nvFatBinSegment`: the header for FatBinary

Fatbinary

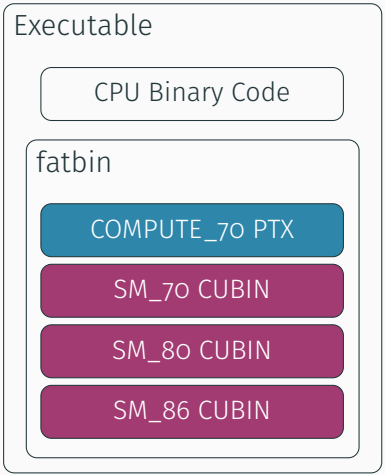
Fatbinary for CUDA has:



Fatbinary

Fatbinary for CUDA has:

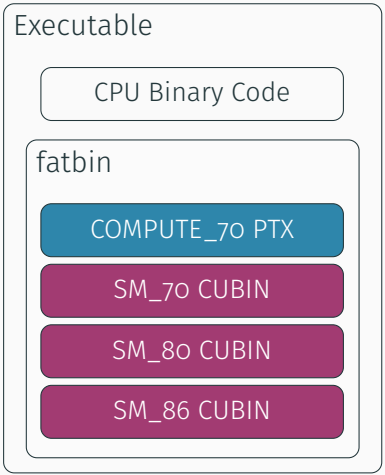
- metadata (magic, versions, info for caching)



Fatbinary

Fatbinary for CUDA has:

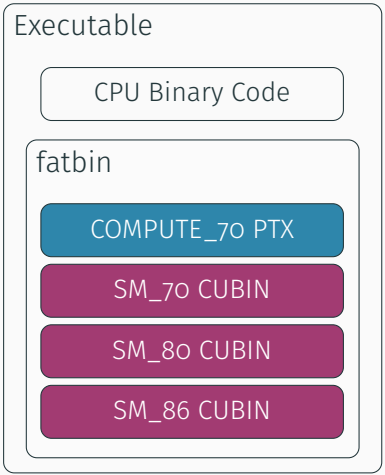
- metadata (magic, versions, info for caching)
- one representation in **PTX** per each version targeted



Fatbinary

Fatbinary for CUDA has:

- metadata (magic, versions, info for caching)
- one representation in **PTX** per each version targeted
- one representation in **SASS** per each architecture targeted



PTX: Parallel Thread Execution

- PTX is a text-based compilation artifact

PTX: Parallel Thread Execution

- PTX is a text-based compilation artifact
- represents the code of one or more kernels

PTX: Parallel Thread Execution

- PTX is a text-based compilation artifact
- represents the code of one or more kernels
- optionally stored compressed (LZ77)

PTX: Parallel Thread Execution

- PTX is a text-based compilation artifact
- represents the code of one or more kernels
- optionally stored compressed (LZ77)

PTX virtual architecture:

- usual load/store RISC-like architecture

PTX: Parallel Thread Execution

- PTX is a text-based compilation artifact
- represents the code of one or more kernels
- optionally stored compressed (LZ77)

PTX virtual architecture:

- usual load/store RISC-like architecture
- unlimited virtual registers

PTX: Parallel Thread Execution

- PTX is a text-based compilation artifact
- represents the code of one or more kernels
- optionally stored compressed (LZ77)

PTX virtual architecture:

- usual load/store RISC-like architecture
- unlimited virtual registers
- no concept of stack

PTX: Parallel Thread Execution

- PTX is a text-based compilation artifact
- represents the code of one or more kernels
- optionally stored compressed (LZ77)

PTX virtual architecture:

- usual load/store RISC-like architecture
- unlimited virtual registers
- no concept of stack
- every instruction can be predicated

PTX format

PTX format

- PTX is composed by an header, kernel and functions entries

PTX format

- PTX is composed by an header, kernel and functions entries
- PTX header defines a couple of important information
  - the PTX version targeted
  - the architecture that PTX should be translated to
  - the size of pointers

```
.version 8.0
.target sm_52
.address_size 64
```

PTX functions

Every PTX has one or more kernel and functions entry:

```
.visible .entry matrix_multiplication(
    .param .u64 param_0,
    .param .u64 param_1,
    .param .u32 param_2,
    .param .u64 param_3
) {

    .reg .pred %p<6>;
    .reg .b32 %r<59>;
    .reg .b64 %rd<35>;

    ld.param.u64 %rd18, [param_0];
    mov.u32 %r21, %ntid.y;
    mov.u32 %r22, %ctaid.y;

    ...
}
```

PTX functions

Every PTX has one or more kernel and functions entry:

- the signature (parameters)

```
.visible .entry matrix_multiplication(
    .param .u64 param_0,
    .param .u64 param_1,
    .param .u32 param_2,
    .param .u64 param_3
) {

    .reg .pred %p<6>;
    .reg .b32 %r<59>;
    .reg .b64 %rd<35>;

    ld.param.u64 %rd18, [_param_0];
    mov.u32 %r21, %ntid.y;
    mov.u32 %r22, %ctaid.y;

    ...
}
```

PTX functions

Every PTX has one or more kernel and functions entry:

- the signature (parameters)
- declaration of registers

```
.visible .entry matrix_multiplication(
    .param .u64 param_0,
    .param .u64 param_1,
    .param .u32 param_2,
    .param .u64 param_3
) {

    .reg .pred %p<6>;
    .reg .b32 %r<59>;
    .reg .b64 %rd<35>;

    ld.param.u64 %rd18, [param_0];
    mov.u32 %r21, %ntid.y;
    mov.u32 %r22, %ctaid.y;

    ...
}
```

PTX functions

Every PTX has one or more kernel and functions entry:

- the signature (parameters)
- declaration of registers
- virtual instructions

```
.visible .entry matrix_multiplication(
    .param .u64 param_0,
    .param .u64 param_1,
    .param .u32 param_2,
    .param .u64 param_3
) {

    .reg .pred %p<6>;
    .reg .b32 %r<59>;
    .reg .b64 %rd<35>;

    ld.param.u64 %rd18, [param_0];
    mov.u32 %r21, %ntid.y;
    mov.u32 %r22, %ctaid.y;

    ...
}
```

PTX instruction

PTX instruction

The usual instruction is:

```
opcode.type.modifier operand0, operand1, operand2, operand3;
```

PTX instruction

The usual instruction is:

```
opcode.type.modifier operand0, operand1, operand2, operand3;
```

```
st.shared.u32 %r25, %r15
```

PTX instruction

The usual instruction is:

```
opcode.type.modifier operand0, operand1, operand2, operand3;
```

```
st.shared.u32 %r25, %r15
```

Every statement already gives a lot of information:

- type information
- kind of memory loaded or stored
- possible modifiers (saturation, zero-extension ...)

PTX operands

Some operands include:

- immediate values
- registers:
  - various ”general purpose” registers with different sizes (1, 4, 8, 16, 32, 64, 128)
  - predicate registers: 1 bit register
  - special registers: read only (e.g. `tid.x`)
- memory, addressing by:
  - an immediate `[0x0]`
  - a register `[%rd5]`
  - a label `[.lbl]`
  - a combination

Set of PTX opcodes

Category	Opcode examples
Arithmetic	`add`, `mul`, `mad`
Logical / bitwise	`and`, `or`, `xor`
Comparison	`set`, `selp`
Control flow	`bra`, `call`, `ret`
Memory access	`ld`, `st`
Type conversion	`cvt`
Synchronization	`bar.sync`

How to translate those?

PTX public documentation

Parallel Thread Execution ISA Version 9.1

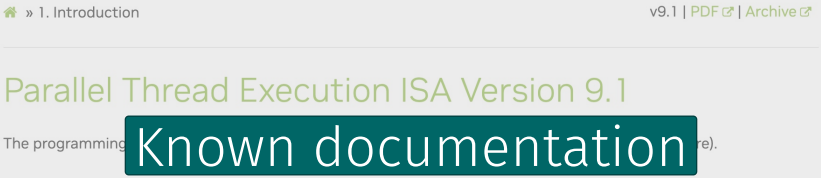
The programming guide to using PTX (Parallel Thread Execution) and ISA (Instruction Set Architecture).

1. Introduction

This document describes PTX, a low-level *parallel thread execution* virtual machine and instruction set architecture (ISA). PTX exposes the GPU as a data-parallel computing *device*.

`https://docs.nvidia.com/cuda/parallel-thread-execution/`

PTX public documentation



https://docs.nvidia.com/cuda/parallel-thread-execution/

Translating to IL

```
mov.u32 %r25, %tid.x;
```

Translating to IL

```
mov.u32 %r25, %tid.x;
```

Translating to IL

```
mov.u32 %r25, %tid.x;
```

```
... = ...
```

Translating to IL

```
mov.u32 %r25, %tid.x;
```

```
... = ...
```

Translating to IL

```
mov.u32 %r25, %tid.x;
```

```
... = ... (32 bits)
```

Translating to IL

```
mov.u32 %r25, %tid.x;
```

```
... = ... (32 bits)
```

Translating to IL

```
mov.u32 %r25, %tid.x;
```

```
r25 = ... (32 bits)
```

Translating to IL

```
mov.u32 %r25, %tid.x;
```

```
r25 = ... (32 bits)
```

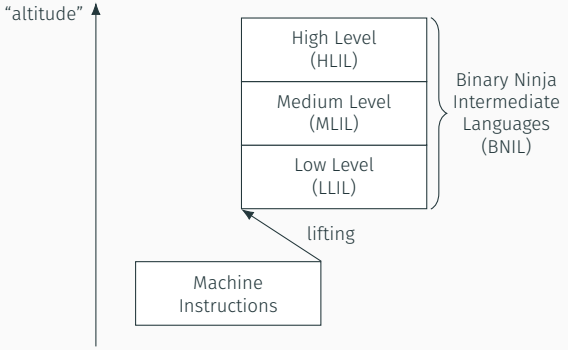
Translating to IL

```
mov.u32 %r25, %tid.x;
```

```
r25 = tid.x (32 bits)
```

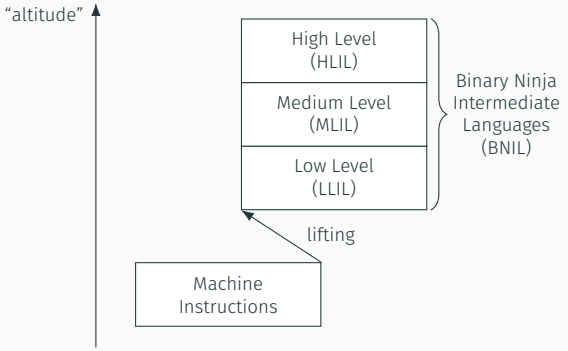
Under Binary Ninja

Under Binary Ninja



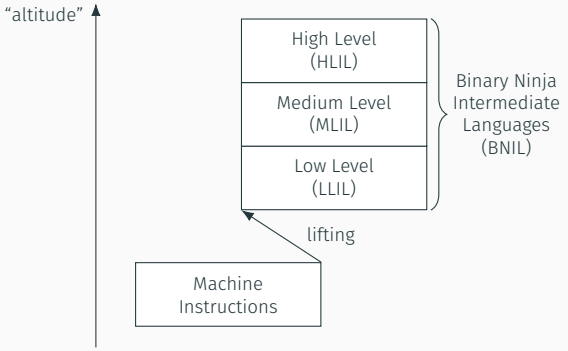
Under Binary Ninja

- idea: we can re-use analyses provided by decompilers



Under Binary Ninja

- idea: we can re-use analyses provided by decompilers
- we need to implement a lifter



ptxNinja

Author: Nicolò Altamura

Explore PTX code through Binary Ninja

Description

ptxNinja is an architecture plugin for Binary Ninja targeting PTX, the language for the virtual architecture of CUDA-based GPU. This allows you to:

- explore PTX binaries integrating analyses already available in the wild
- navigate through GPU kernels and functions
- integrate PTX into your existing automated tools thanks to Binary Ninja

https://github.com/seekbytes/ptxNinja

```
int32_t ntid.x
int32_t r2 = ctaid.x * ntid.x + tid.x
int32_t f24 = 0

if (param_2 s>= 1)
    int32_t i_1 = param_2 & 3
    f24 = 0
    int32_t r44_1 = 0

    if (param_2 - 1 s>= 3)
        int32_t i = param_2 - i_1
        int32_t* rd32_1 = param_1 + zx.q(r2) * 4
        int32_t* rd31_1 = param_0 + zx.q(param_2 * r1) * 4 + 8
        int32_t r23_1 = *rd32_1

        do
            int32_t* rd23_1 = &rd32_1[zx.q(param_2)]
            int32_t* rd24_1 = &rd23_1[zx.q(param_2)]
            int32_t* rd25_1 = &rd24_1[zx.q(param_2)]
            rd32_1 = &rd25_1[zx.q(param_2)]
            f24 = f24 + r23_1 * rd31_1[-2] + *rd23_1 * rd31_1[-1]
                + *rd24_1 * *rd31_1 + *rd25_1 * rd31_1[1]
            r44_1 += 4
            rd31_1 = &rd31_1[4]
            i -= 4
        while (i != 0)

        arg2 = i_1 == 0

    if (not(arg2))
        int32_t* rd34_1 = param_1 + zx.q(r44_1 * param_2 + r2) * 4
        int32_t* rd33_1 = param_0 + zx.q(param_2 * r1 + r44_1) * 4

        do
```

```
    int32_t ntid.x
    int32_t r2 = ctaid.x * ntid.x + tid.x
    int32_t f24 = 0

    if (param_2 s>= 1)
        int32_t i_1 = param_2 & 3
        f24 = 0
        int32_t r44_1 = 0

        if (param_2 - 1 s>= 3)
            int32_t i = param_2 - i_1
            int32_t* rd32_1 = param_1 + zx.q(r2) * 4
            int32_t* rd31_1 = param_0 + zx.q(param_2 * r1) * 4 + 8
            int32_t r23_1 = *rd32_1

            do
                int32_t* rd23_1 = &rd32_1[zx.q(param_2)]
                int32_t* rd24_1 = &rd23_1[zx.q(param_2)]
                int32_t* rd25_1 = &rd24_1[zx.q(param_2)]
                rd32_1 = &rd25_1[zx.q(param_2)]
                f24 = f24 + r23_1 * rd31_1[-2] + *rd23_1 * rd31_1[-1]
                    + *rd24_1 * *rd31_1 + *rd25_1 * rd31_1[1]
                r44_1 += 4
                rd31_1 = &rd31_1[4]
                i -= 4
            while (i != 0)

            arg2 = i_1 == 0

        if (not(arg2))
            int32_t* rd34_1 = param_1 + zx.q(r44_1 * param_2 + r2) * 4
            int32_t* rd33_1 = param_0 + zx.q(param_2 * r1 + r44_1) * 4
```

```
int32_t ntid.x
int32_t r2 = ctaid.x * ntid.x + tid.x
int32_t f24 = 0

if (param_2 s>= 1)
    int32_t i_1 = param_2 & 3
    f24 = 0
    int32_t r44_1 = 0

    if (param_2 - 1 s>= 3)
        int32_t i = param_2 - i_1
        int32_t* rd32_1 = param_1 + zx.q(r2) * 4
        int32_t* rd31_1 = param_0 + zx.q(param_2 * r1) * 4 + 8
        int32_t r23_1 = *rd32_1

        do
            i
            i
            int32_t* rd25_1 = &rd24_1[zx.q(param_2)]
            rd32_1 = &rd25_1[zx.q(param_2)]
            f24 = f24 + r23_1 * rd31_1[-2] + *rd23_1 * rd31_1[-1]
                + *rd24_1 * *rd31_1 + *rd25_1 * rd31_1[1]
            r44_1 += 4
            rd31_1 = &rd31_1[4]
            i -= 4
        while (i != 0)

        arg2 = i_1 == 0

    if (not(arg2))
        int32_t* rd34_1 = param_1 + zx.q(r44_1 * param_2 + r2) * 4
        int32_t* rd33_1 = param_0 + zx.q(param_2 * r1 + r44_1) * 4
```

Matrix multiplication

```
int32_t r54 = param_7 i* param_6
int64_t rd5 = sx.q(r54 * (r1 s/ (r54 * param_5) * param_5 + r1 s/ r54 s% param_5))
float f1 = rsqrt(param_7)
bool p2 = param_6 s< 1

if (not(p2))
    int32_t r3_1 = r1 s/ param_7 s% param_6 i* param_7

    if (param_7 s>= 0)
        int32_t r16_1 = param_7 & 3
        int32_t f128_1 = -0x800000
        int32_t r83_1 = 0

        do
            int32_t f127_1 = 0
            int32_t r86_1 = 0

            if (arg1 s>= 3)
                int32_t* rd71_1 = param_1 + (rd5 << 2) + 8
                r86_1 = 0
                int32_t* rd72_1 = param_0 + ((rd5 + sx.q(r3_1)) << 2) + 8
                int32_t i = param_7 i- r16_1
                int32_t f44_1 = rd71_1[-2]

                do
                    f127_1 = rd72_1[1] * rd71_1[1] + *rd72_1 * *rd71_1
                        + rd72_1[-1] * rd71_1[-1] + rd72_1[-2] * f44_1 + f127_1
                    r86_1 += 4
                    rd72_1 = &rd72_1[4]
                    rd71_1 = &rd71_1[4]
                    i -= 4
                while (i != 0)
```

Scale factor $\frac{1}{\sqrt{d_k}}$

```
int64_t rd5 = sx.q(r54 * (r1 s/ (r54 * param_5)
float f1 = rsqrt(param_7)
bool p2 = param_6 s< 1
```

Dot product $Q \cdot K$

```
f127_1 = rd72_1[1] * rd71_1[1] + *rd72_1 * *rd71_1
    + rd72_1[-1] * rd71_1[-1] + rd72_1[-2] * f44_1 + f127_1
r86_1 += 4
```

Scaling + max tracking

```
int32_t f61_1 = f1 i* f127_1
*(&__local_depot0 + arg3) = f61_1

if (f128_1 s<= f61_1)
    f128_1 = f61_1
```

Scale factor $\frac{1}{\sqrt{d_k}}$

```
int64_t rd5 = sx.q(r54 * (r1 s/ (r54 * param_5)
float f1 = rsqrt(param_7)
bool p2 = param_6 s< 1
```

Dot product $Q \cdot K$

```
f127_1 = rd72_1[1] * rd71_1[1] + *rd72_1 * *rd71_1
    + rd72_1[-1] * rd71_1[-1] + rd72_1[-2] * f44_1 + f127_1
r86_1 += 4
```

Transformer attention

Scaling + max tracking

```
int32_t f61_1 = f1 i* f127_1
*(&__local_depot0 + arg3) = f61_1

if (f128_1 s<= f61_1)
    f128_1 = f61_1
```

The end?

- instructions are in clear text and divided into functions
- functions have a correct signature
- language already typed
- PTX is well documented

The end?

- instructions are in clear text and divided into functions
- functions have a correct signature
- language already typed
- PTX is well documented

This solves part of reverse engineering problems

Challenges still remain...

However:

- what about complex types? control flow structure?
- no reasoning about dataflow
- PTX hides micro-architectural details
- SASS: still unknown
- several kernels are distributed only as a set of SASS executables

Challenges still remain...

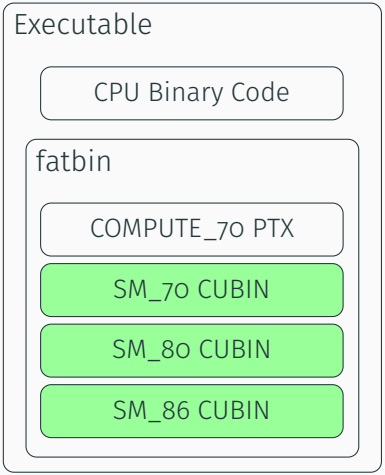
However:

- what about complex types? control flow structure?
- no reasoning about dataflow
- PTX hides micro-architectural details
- SASS: still unknown
- several kernels are distributed only as a set of SASS executables

Let's move to decompilation of SASS

CUbin

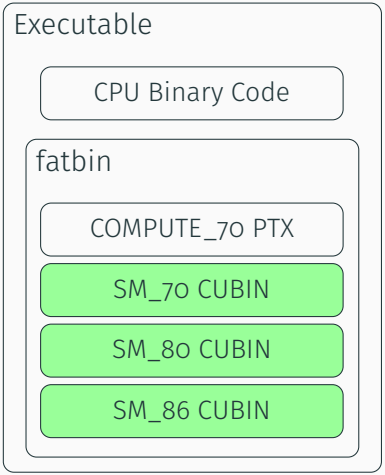
The other format of fatbinary: **CUbin**



CUbin

The other format of fatbinary: **CUbin**

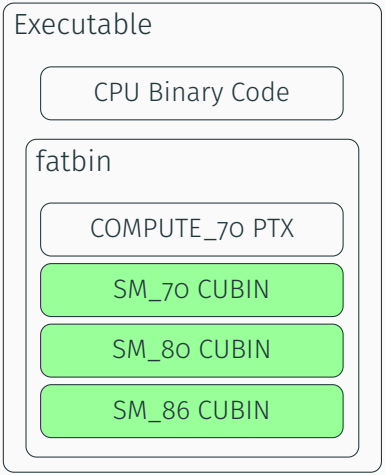
- ELF file with machine `0xbe` (EM_CUDA)



CUbin

The other format of fatbinary: **CUbin**

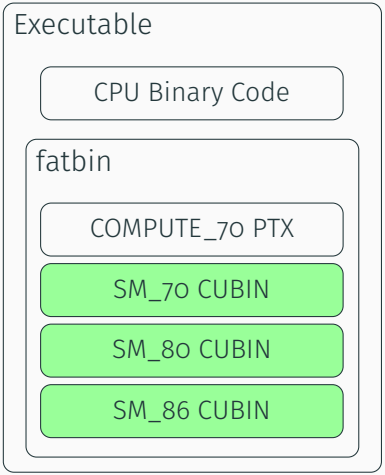
- ELF file with machine `0xbe` (EM_CUDA)
- similar to native executable of CPU code



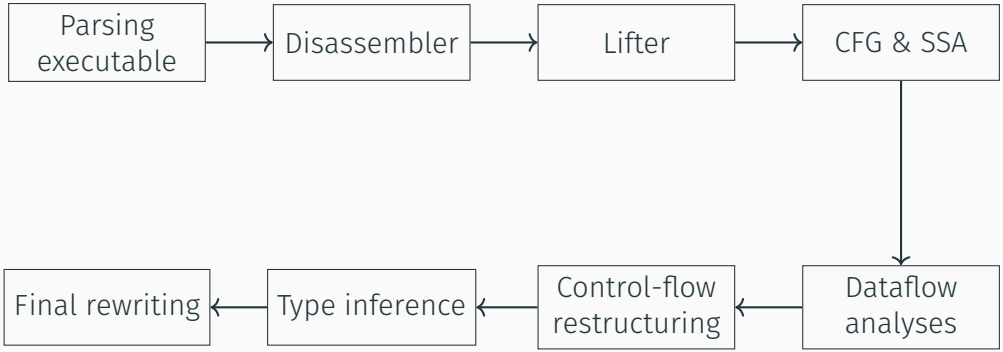
CUbin

The other format of fatbinary: **CUbin**

- ELF file with machine `0xbe` (EM_CUDA)
- similar to native executable of CPU code
- sections for CUDA:
  - executable sections: contains SASS instructions
  - `.nv.info`: includes some specific info for CUDA



The decompilation pipeline



Disassembler

Goal: extracting the instructions from raw bytes

Disassembler

Goal: extracting the instructions from raw bytes

- NVIDIA provide already disassemblers (`cuobjdump`)

Disassembler

Goal: extracting the instructions from raw bytes

- NVIDIA provide already disassemblers (`cuobjdump`)
- black-box binaries
- need time and resources to decompile them

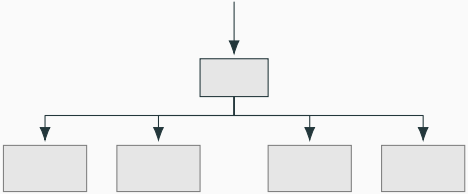
Disassembler

Goal: extracting the instructions from raw bytes

- NVIDIA provide already disassemblers (`cuobjdump`)
- black-box binaries
- need time and resources to decompile them

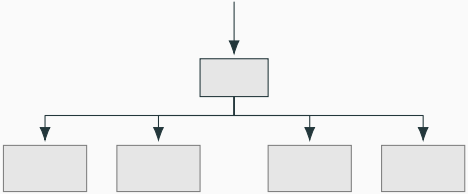
```
/*0000*/ MOV R1, c[0x0][0x28] ;                /* 0x00000a0000017a02 */
                                               /* 0x000fc40000000f00 */
/*0010*/ @!PT SHFL.IDX PT, RZ, RZ, RZ, RZ ;    /* 0x000000ffffff389 */
                                               /* 0x000fe200000e00ff */
```

Finding the instruction specification

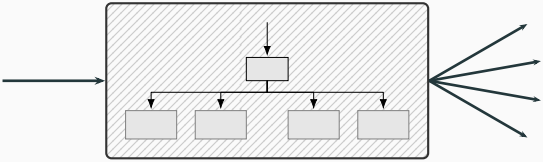


Reverse engineering `nvdisasm`

Finding the instruction specification

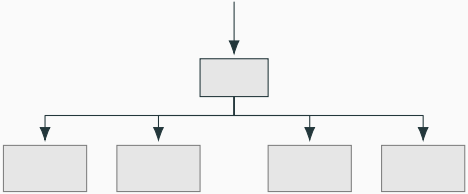


Reverse engineering `nvdisasm`

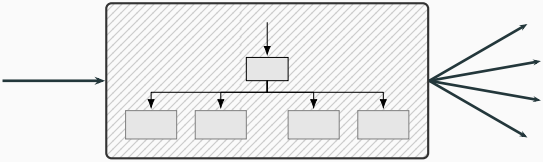


Treat `nvdisasm` as oracle

Finding the instruction specification



Reverse engineering `nvdisasm`



Treat `nvdisasm` as oracle

Studying the raw instructions

```
02 7A 01 00 00 0A 00 00 00 0F 00 00 00 C4 0F 00 89 F3 FF FF FF 00 00 00 FF
00 0E 00 00 E2 0F 00 19 79 06 00 00 00 00 00 00 25 00 00 00 28 0E 00 19 79
03 00 00 00 00 00 00 21 00 00 00 24 0E 00 24 7A 06 06 00 00 00 00 03 02 8E
07 00 CA 1F 00 0C 7A 00 06 00 5E 00 00 70 62 F0 03 00 D8 0F 00 4D 09 00 00
```

- how many bytes per instruction?

Studying the raw instructions

```
02 7A 01 00 00 0A 00 00 00 0F 00 00 00 C4 0F 00 89 F3 FF FF FF 00 00 00 FF
00 0E 00 00 E2 0F 00 19 79 06 00 00 00 00 00 00 25 00 00 00 28 0E 00 19 79
03 00 00 00 00 00 00 21 00 00 00 24 0E 00 24 7A 06 06 00 00 00 00 03 02 8E
07 00 CA 1F 00 0C 7A 00 06 00 5E 00 00 70 62 F0 03 00 D8 0F 00 4D 09 00 00
```

- how many bytes per instruction?
- fixed-size instruction?

Studying the raw instructions

```
02 7A 01 00 00 0A 00 00 00 0F 00 00 00 C4 0F 00 89 F3 FF FF FF 00 00 00 FF
00 0E 00 00 E2 0F 00 19 79 06 00 00 00 00 00 00 25 00 00 00 28 0E 00 19 79
03 00 00 00 00 00 00 21 00 00 00 24 0E 00 24 7A 06 06 00 00 00 00 03 02 8E
07 00 CA 1F 00 0C 7A 00 06 00 5E 00 00 70 62 F0 03 00 D8 0F 00 4D 09 00 00
```

- how many bytes per instruction?
- fixed-size instruction?
- is context-dependent?

Binary diffing

Idea: place the same instruction in different places

Binary diffing

Idea: place the same instruction in different places

```
...
int a = b + c;
int idx = blockIdx.x;
int idy = blockIdx.y;
...
```


```
10 44 56 07 04 05
 ↪  aa 08 9e 95
 ↪  12 15 19 43
 ↪  25 a6
```

Binary diffing

Idea: place the same instruction in different places

```
...
int a = b + c;
int idx = blockIdx.x;
int idy = blockIdx.y;
...
```

nvcc

```
10 44 56 07 04 05
↪  aa 08 9e 95
↪  12 15 19 43
↪  25 a6
```

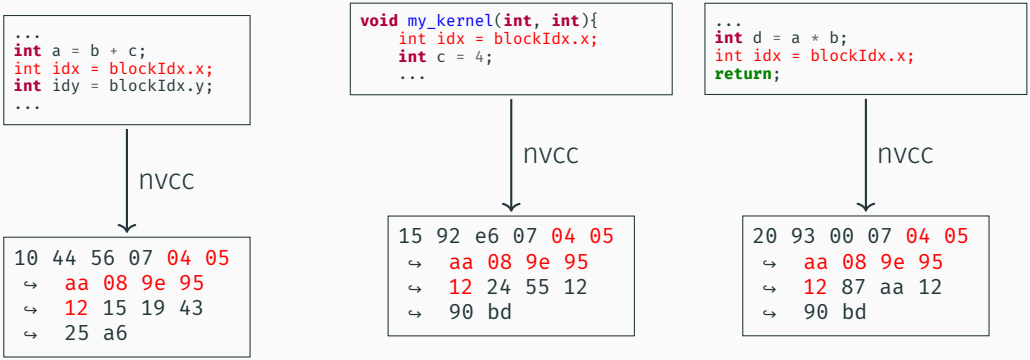
```
void my_kernel(int, int){
    int idx = blockIdx.x;
    int c = 4;
    ...
```

nvcc

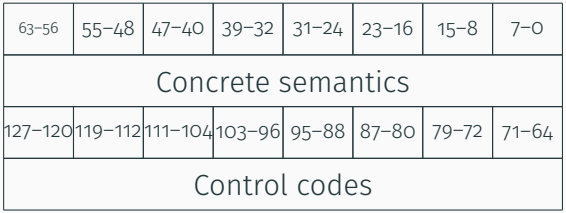
```
15 92 e6 07 04 05
↪  aa 08 9e 95
↪  12 24 55 12
↪  90 bd
```

Binary diffing

Idea: place the same instruction in different places



Anatomy of SASS instruction (Volta)



- 16 bytes for every instruction
- 8 bytes for the concrete instruction
- 8 bytes for context-sensitive information

Anatomy of Control Codes (Volta)

64–18	17–11	10–8	7–5	4	3–0
unusued	waitbar	read-bar	write-bar	–	stall

- defines how the pipeline is configured for this instruction
- **out of our discussion**: the disassembler will simply skip this information

How to mine SASS instructions

How to mine SASS instructions

Goal: Enumerate most of instructions and synthesize the instruction layout

How to mine SASS instructions

Goal: Enumerate most of instructions and synthesize the instruction layout

1. create the smallest sass binary

How to mine SASS instructions

Goal: Enumerate most of instructions and synthesize the instruction layout

1. create the smallest sass binary
2. edit the first instruction

How to mine SASS instructions

Goal: Enumerate most of instructions and synthesize the instruction layout

1. create the smallest sass binary
2. edit the first instruction
3. dump the instruction text via `nvdisasm`

How to mine SASS instructions

Goal: Enumerate most of instructions and synthesize the instruction layout

1. create the smallest sass binary
2. edit the first instruction
3. dump the instruction text via `nvdisasm`
4. mutate the bytes buffer and annotate the difference

How to mine SASS instructions

Goal: Enumerate most of instructions and synthesize the instruction layout

1. create the smallest sass binary
2. edit the first instruction
3. dump the instruction text via `nvdisasm`
4. mutate the bytes buffer and annotate the difference
5. repeat from 1.

How to mine SASS instructions

Goal: Enumerate most of instructions and synthesize the instruction layout

1. create the smallest sass binary
2. edit the first instruction
3. dump the instruction text via `nvdisasm`
4. mutate the bytes buffer and annotate the difference
5. repeat from 1.

Ideally, we want to have the most information with the fewest change.

Mining SASS instructions

```
[02 7a 01 00 00 0a 00
↪  00]
```

MOV R1,
c[0x0][0x28]

Mining SASS instructions

```
[02 7a 01 00 00 0a 00
↪  00]
```

MOV R1,
c[0x0][0x28]

```
[02 7a 01 02 00 0a 00
↪  00]
```

Mining SASS instructions

```
[02 7a 01 00 00 0a 00
↪  00]
```

MOV R1,
c[0x0][0x28]

```
[02 7a 01 02 00 0a 00
↪  00]
```

P2R R1, PR, R0,
c[0x0][0x28]

Mining SASS instructions

```
[02 7a 01 00 00 0a 00
↪  00]
```

MOV R1,
c[0x0][0x28]

```
[02 7a 01 02 00 0a 00
↪  00]
```

P2R R1, PR, R0,
c[0x0][0x28]

New opcode!

Mining SASS instructions

```
[02 7a 01 00 00 0a 00
↪  00]
```

MOV R1, c[0x0][0x28]

```
[02 7a 01 02 00 0a 00
↪  00]
```

P2R R1, PR, R0, c[0x0][0x28]

New opcode!

```
[02 7a 01 00 03 0a 00
↪  00]
```

Mining SASS instructions

```
[02 7a 01 00 00 0a 00
↪  00]
```

MOV R1, c[0x0][0x28]

```
[02 7a 01 02 00 0a 00
↪  00]
```

P2R R1, PR, R0, c[0x0][0x28]

New opcode!

```
[02 7a 01 00 03 0a 00
↪  00]
```

Invalid

Mining SASS instructions

```
[02 7a 01 00 00 0a 00
↪  00]
```

MOV R1, c[0x0][0x28]

```
[02 7a 01 02 00 0a 00
↪  00]
```

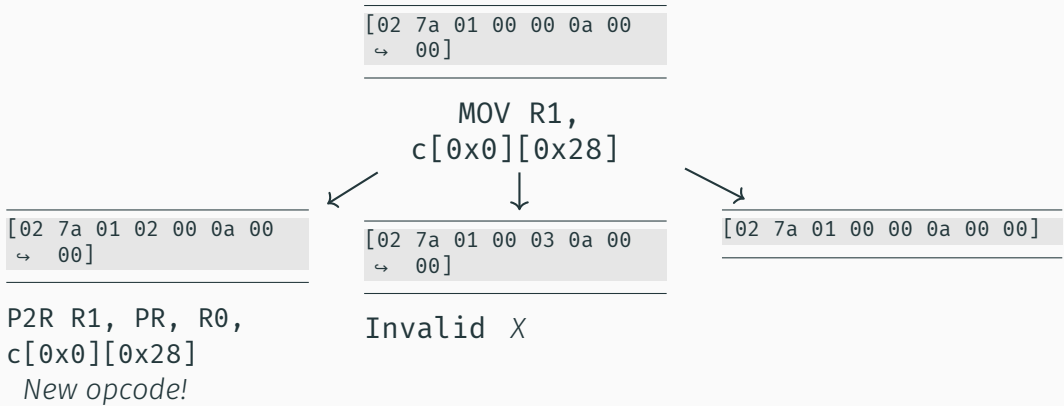
P2R R1, PR, R0, c[0x0][0x28]

New opcode!

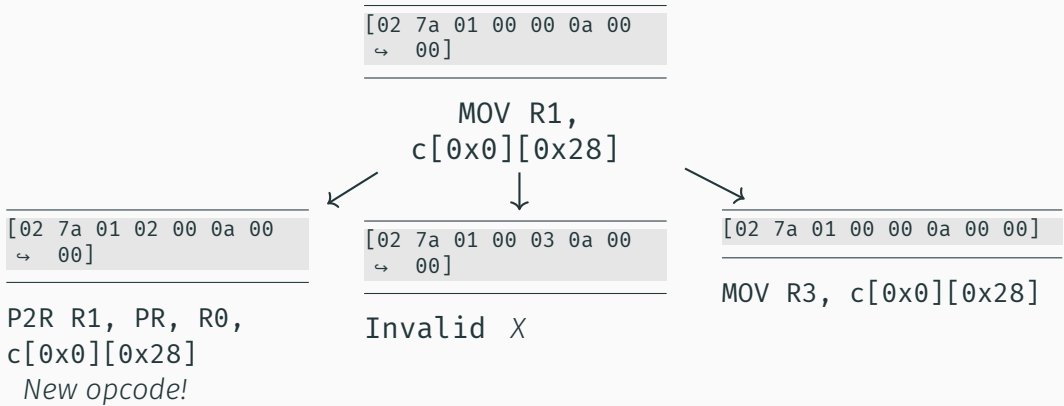
```
[02 7a 01 00 03 0a 00
↪  00]
```

Invalid ✗

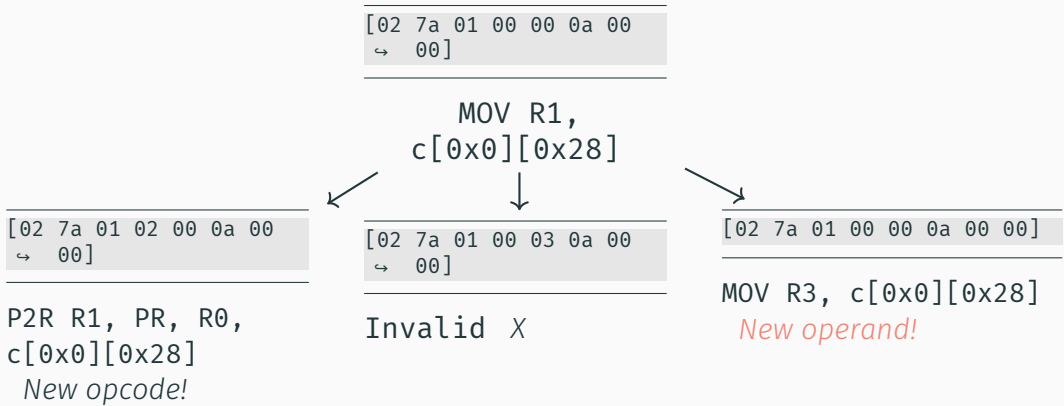
Mining SASS instructions



Mining SASS instructions



Mining SASS instructions



Opcode and operands

Thanks to enumeration of instructions:

- Lots of more opcodes compared to PTX
- Several operands might be identified

Operand	Example
Immediate	`0x0`, `0x10`
Registers	`RZ`, `R0`, `R1`
Uniform registers	`Rd0`, `Rd1`
Predicates	`P1`, `P2`
Constant Bank	`c[0x0][0x168]`
Special Register	`TID.X, CTAID.Y`
Memory	`[R2]`, `[R18+0x140]`

Every operand can be negative, positive, or absolute.

Lifter

- assembly instructions are helpful for humans

Lifter

- assembly instructions are helpful for humans
- for the next steps we need to reason about the semantics

Lifter

- assembly instructions are helpful for humans
- for the next steps we need to reason about the semantics
- concrete semantics can be derived by intuition...

Lifter

- assembly instructions are helpful for humans
- for the next steps we need to reason about the semantics
- concrete semantics can be derived by intuition...

```
MOV R21, RZ → r21 := 0
```

Lifter

- assembly instructions are helpful for humans
- for the next steps we need to reason about the semantics
- concrete semantics can be derived by intuition...

`MOV R21, RZ` → `r21 := 0`

but what about `imad r3, r3, 0x10, r2`?

Lifter

- assembly instructions are helpful for humans
- for the next steps we need to reason about the semantics
- concrete semantics can be derived by intuition...

`MOV R21, RZ` → `r21 := 0`

but what about `imad r3, r3, 0x10, r2`?

This does not scale!

Synthesis of concrete semantics

- NVIDIA provides debugging tools for CUDA (`cuda-gdb`)
- group readable instructions by opcode (sort alphabetically)
- ignore pipeline bytes

Obtaining traces

1. feed the initial state
2. breakpoint on the instruction
3. single step on the instruction
4. obtain the single trace
5. synthesize a possible semantics
6. reset and mutate the initial state and repeat from 1.

Idea: take the intersection of all possible semantics for that instruction.

Synthesis from initial state

```
IADD R1, R2, R3;
```

Run 1

```
R1 = 0x0
R2 = 0x04
R3 = 0x01
```

Synthesis from initial state

```
IADD R1, R2, R3;
```

Run 1

```
R1 = 0x0
R2 = 0x04
R3 = 0x01
```

Final

```
R1 = 0x05
R2 = 0x04
R3 = 0x01
```

Synthesis from initial state

```
IADD R1, R2, R3;
```

Run 1

```
R1 = 0x0
R2 = 0x04
R3 = 0x01
```

Final

```
R1 = 0x05
R2 = 0x04
R3 = 0x01
```

```
R1 = R2 + R3
```

Synthesis from initial state

```
IADD R1, R2, R3;
```

Run 1

```
R1 = 0x0
R2 = 0x04
R3 = 0x01
```

Final

```
R1 = 0x05
R2 = 0x04
R3 = 0x01
```

```
R1 = R2 + R3
```

Run 2

```
R1 = 0x10
R2 = 0x04
R3 = 0x06
```

Synthesis from initial state

```
IADD R1, R2, R3;
```

Run 1

```
R1 = 0x0
R2 = 0x04
R3 = 0x01
```

Final

```
R1 = 0x05
R2 = 0x04
R3 = 0x01
```

Run 2

```
R1 = 0x10
R2 = 0x04
R3 = 0x06
```

Final

```
R1 = 0x10
R2 = 0x04
R3 = 0x06
```

```
R1 = R2 + R3
```

Synthesis from initial state

```
IADD R1, R2, R3;
```

Run 1

```
R1 = 0x0
R2 = 0x04
R3 = 0x01
```

Final

```
R1 = 0x05
R2 = 0x04
R3 = 0x01
```

```
R1 = R2 + R3
```

Run 2

```
R1 = 0x10
R2 = 0x04
R3 = 0x06
```

Final

```
R1 = 0x10
R2 = 0x04
R3 = 0x06
```

```
no effect
```

Synthesis from initial state

```
IADD R1, R2, R3;
```

Run 1

```
R1 = 0x0
R2 = 0x04
R3 = 0x01
```

Final

```
R1 = 0x05
R2 = 0x04
R3 = 0x01
```

```
R1 = R2 + R3
```

Run 2

```
R1 = 0x10
R2 = 0x04
R3 = 0x06
```

Final

```
R1 = 0x10
R2 = 0x04
R3 = 0x06
```

```
no effect
```

Run 3

```
R1 = 0x05
R2 = 0x03
R3 = 0x04
```

Synthesis from initial state

```
IADD R1, R2, R3;
```

Run 1

```
R1 = 0x0
R2 = 0x04
R3 = 0x01
```

Final

```
R1 = 0x05
R2 = 0x04
R3 = 0x01
```

`R1 = R2 + R3`

Run 2

```
R1 = 0x10
R2 = 0x04
R3 = 0x06
```

Final

```
R1 = 0x10
R2 = 0x04
R3 = 0x06
```

`no effect`

Run 3

```
R1 = 0x05
R2 = 0x03
R3 = 0x04
```

Final

```
R1 = 0x07
R2 = 0x03
R3 = 0x04
```

Synthesis from initial state

```
IADD R1, R2, R3;
```

Run 1

```
R1 = 0x0
R2 = 0x04
R3 = 0x01
```

Final

```
R1 = 0x05
R2 = 0x04
R3 = 0x01
```

`R1 = R2 + R3`

Run 2

```
R1 = 0x10
R2 = 0x04
R3 = 0x06
```

Final

```
R1 = 0x10
R2 = 0x04
R3 = 0x06
```

`no effect`

Run 3

```
R1 = 0x05
R2 = 0x03
R3 = 0x04
```

Final

```
R1 = 0x07
R2 = 0x03
R3 = 0x04
```

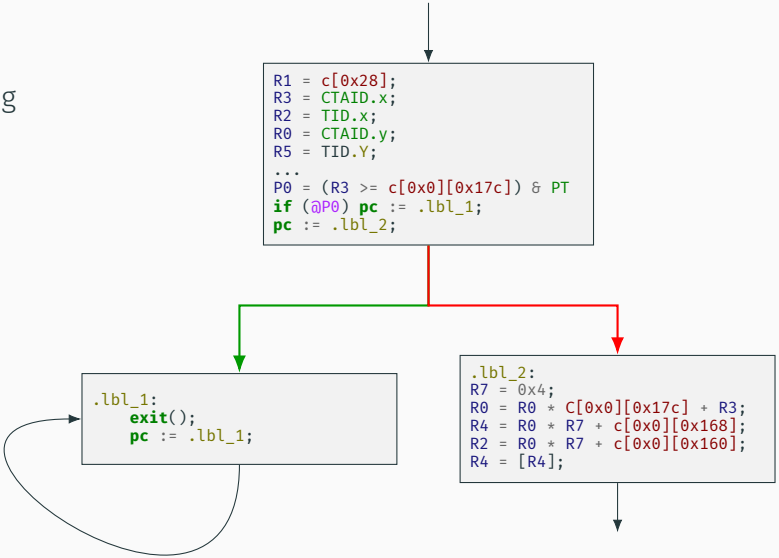
`R1 = R2 + R3`

Easy and hard instructions

- instructions can be predicated
  - if predicate isn't set, the instruction does not change the GPU state
- some instructions are harder to synthesize than others
  - ones that use synchronization
  - ones that use special registers
  - ones that pass thread-information
- some instructions are defined as intrinsics
  - `bar.sync`
  - atomic instructions

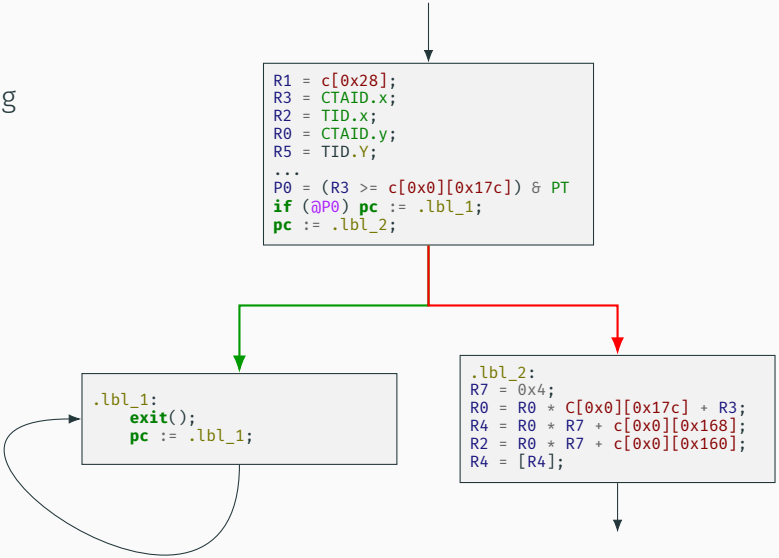
Building of CFG & SSA version

- building basic blocks by grouping instructions



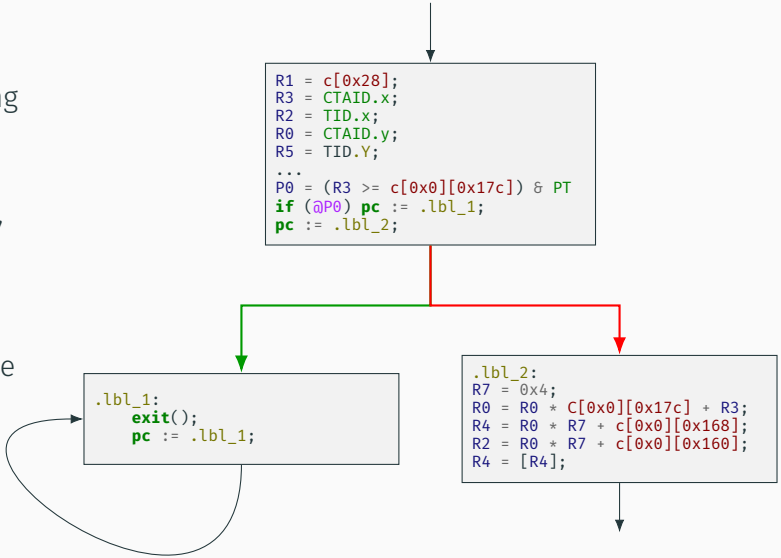
Building of CFG & SSA version

- building basic blocks by grouping instructions
- control-flow instructions: BRA, BRX (`pc := ...`)



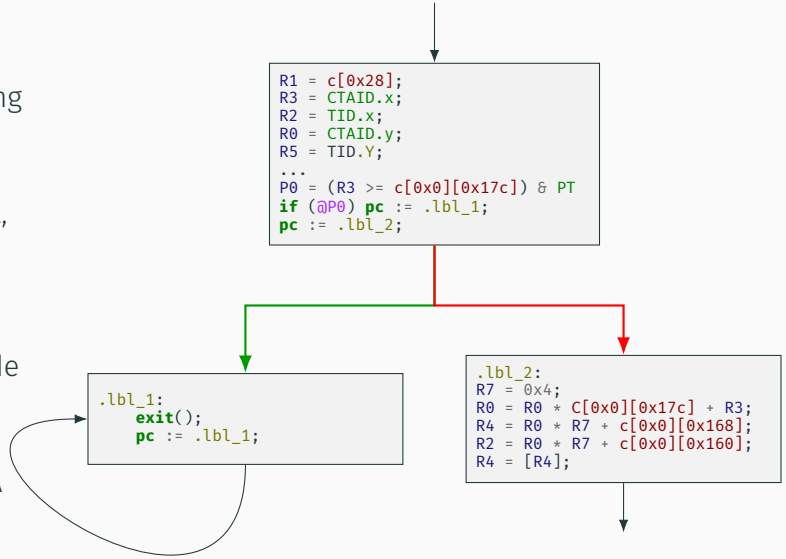
Building of CFG & SSA version

- building basic blocks by grouping instructions
- control-flow instructions: BRA, BRX (`pc := ...`)
- synchronization instructions divide a basic block

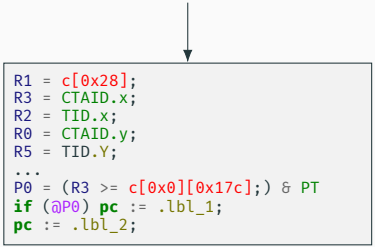


Building of CFG & SSA version

- building basic blocks by grouping instructions
- control-flow instructions: BRA, BRX (`pc := ...`)
- synchronization instructions divide a basic block
- already known algorithm for SSA



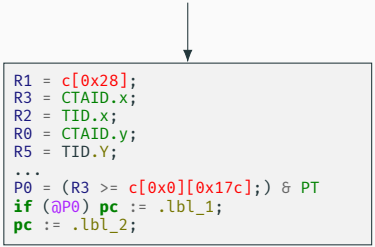
Parameters, and return values



```
R1 = c[0x28];
R3 = CTAID.x;
R2 = TID.x;
R0 = CTAID.y;
R5 = TID.Y;
...
P0 = (R3 >= c[0x0][0x17c];) & PT
if (@P0) pc := .lbl_1;
pc := .lbl_2;
```

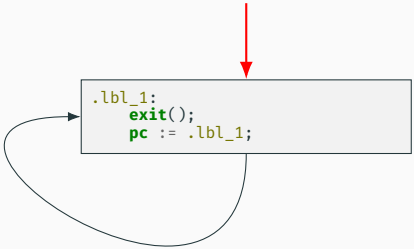
What are c[...]?

Parameters, and return values



```
R1 = c[0x28];
R3 = CTAID.x;
R2 = TID.x;
R0 = CTAID.y;
R5 = TID.Y;
...
P0 = (R3 >= c[0x0][0x17c];) & PT
if (@P0) pc := .lbl_1;
pc := .lbl_2;
```

What are c[...]?



```
.lbl_1:
    exit();
    pc := .lbl_1;
```

Why?

Parameters, and return values

For arguments:

- we can query a specific data structure under `.nv_info`
- type: `EIATTR_KPARAM_INFO`
- size and alignment: `EIATTR_CBANK_PARAM_SIZE`

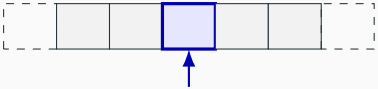
For returns:

- no real ”return”
- `EXIT` instruction to notify the host that kernel ended its task

Dataflow

Goal: track values and simplify the representation

- CUDA paradigm reasons well with 1d-arrays and ***n***-d arrays
- necessary some relational-based analyses




```
// typical recovered pattern
i = ctaid.x * ntid.x + tid.x;

addr = base + i * sizeof(T);
```

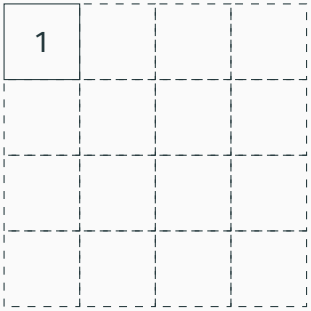
Matrices, and tensors

- hard to recognize
- accessed as raw pointers
- index expressions are the challenge
- range analysis as a solution to understand indexes

```
unknown_A[threadId * 4 + r2]
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

vs.

$A =$ 

Matrices, and tensors

- hard to recognize
- accessed as raw pointers
- index expressions are the challenge
- range analysis as a solution to understand indexes

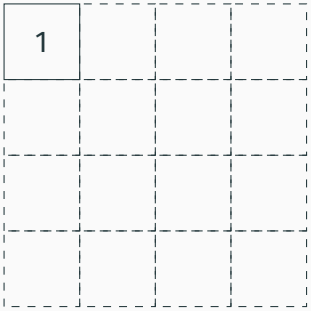
```
unknown_A[threadId * 4 + r2]

threadId = {0, …, 3}
r2 = {0, …, 3}
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

vs.

$A =$



Matrices, and tensors

- hard to recognize
- accessed as raw pointers
- index expressions are the challenge
- range analysis as a solution to understand indexes

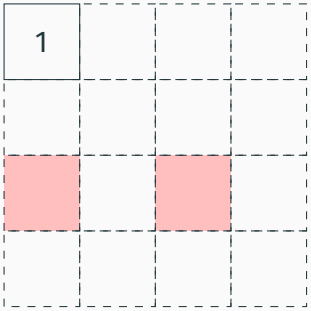
```
unknown_A[threadId * 4 + r2]

threadId = {0, …, 3}
r2 = {0, …, 3}
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

vs.

$A =$



Matrices, and tensors

- hard to recognize
- accessed as raw pointers
- index expressions are the challenge
- range analysis as a solution to understand indexes

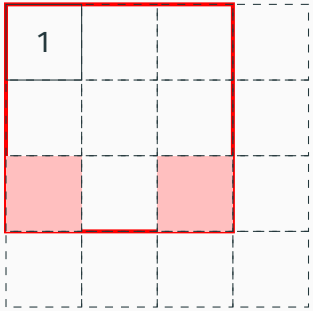
```
unknown_A[threadId * 4 + r2]

threadId = {0, …, 3}
r2 = {0, …, 3}
```

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

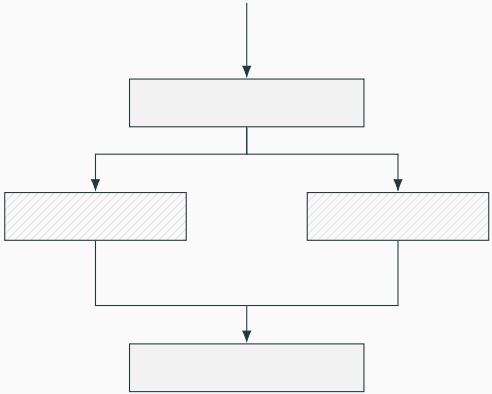
vs.

$A =$



Control-flow restructuring

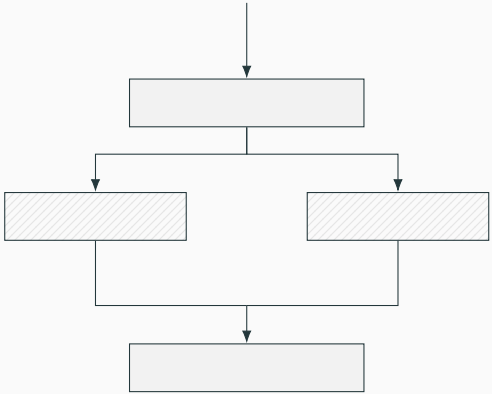
Goal: add the control-flow structures



Control-flow restructuring

Goal: add the control-flow structures

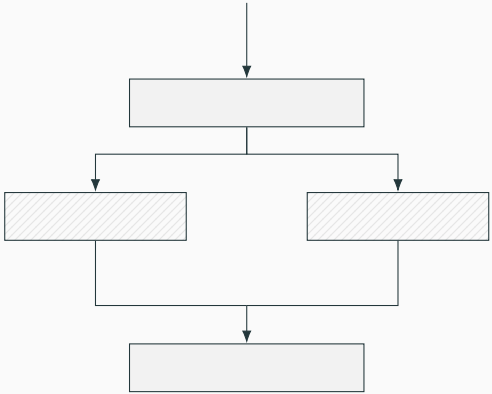
- simple control-flow due to boundaries that CUDA poses to its developers



Control-flow restructuring

Goal: add the control-flow structures

- simple control-flow due to boundaries that CUDA poses to its developers
- the more complex control-flow is, the slower the kernel would be

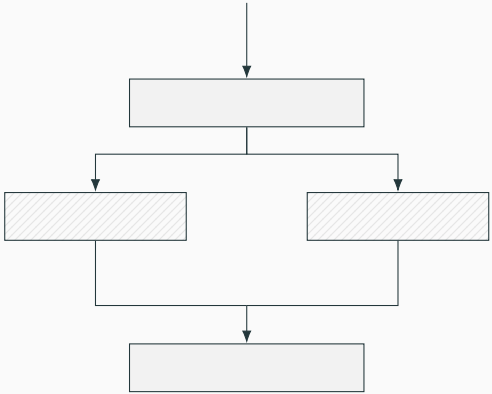


Control-flow restructuring

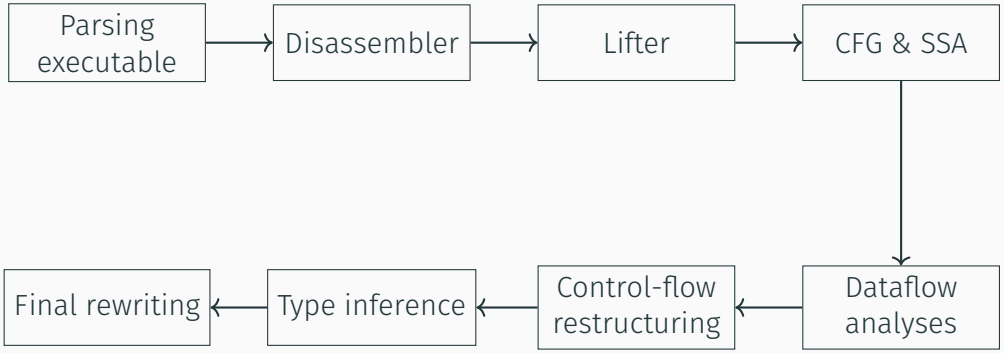
Goal: add the control-flow structures

- simple control-flow due to boundaries that CUDA poses to its developers
- the more complex control-flow is, the slower the kernel would be

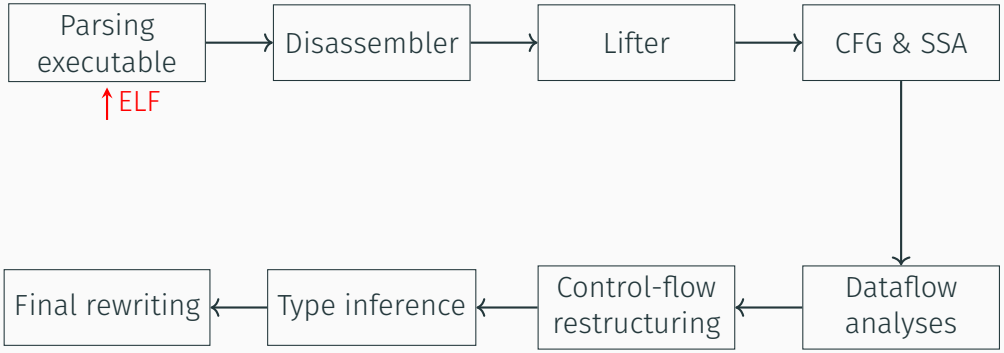
Literature already has plenty of work around this!



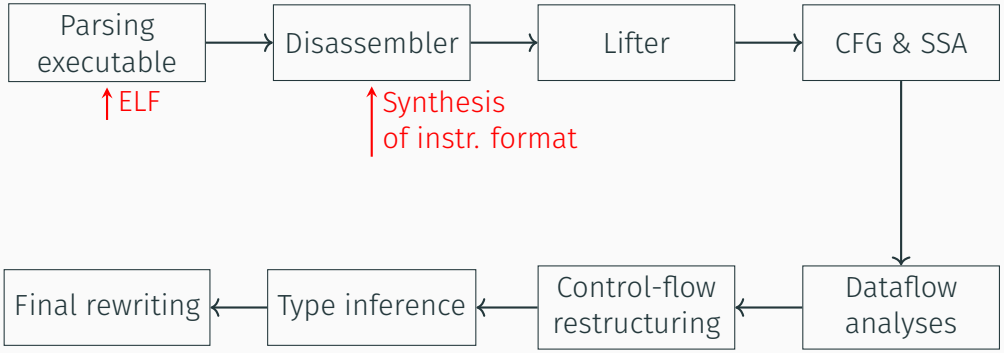
Further research



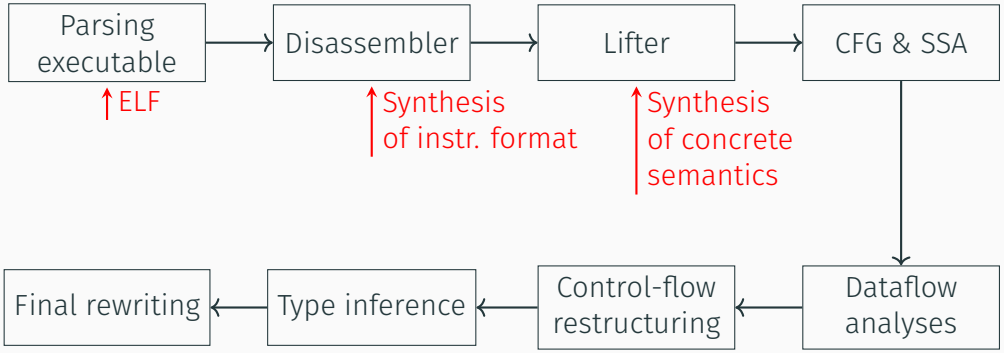
Further research



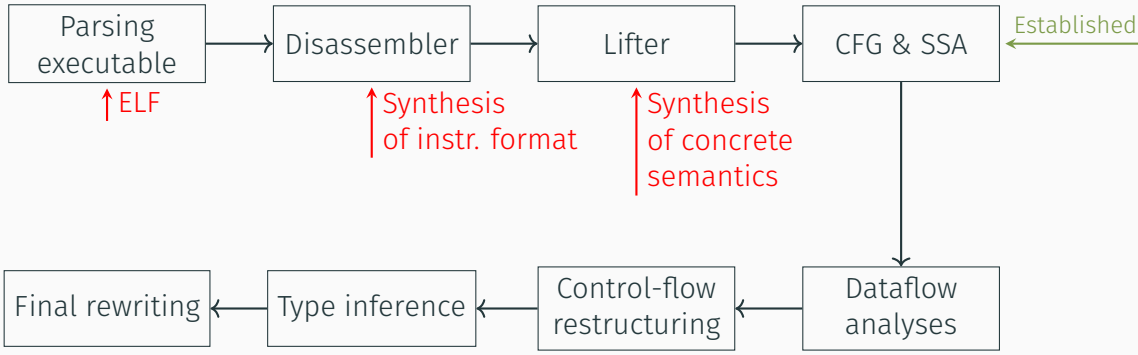
Further research



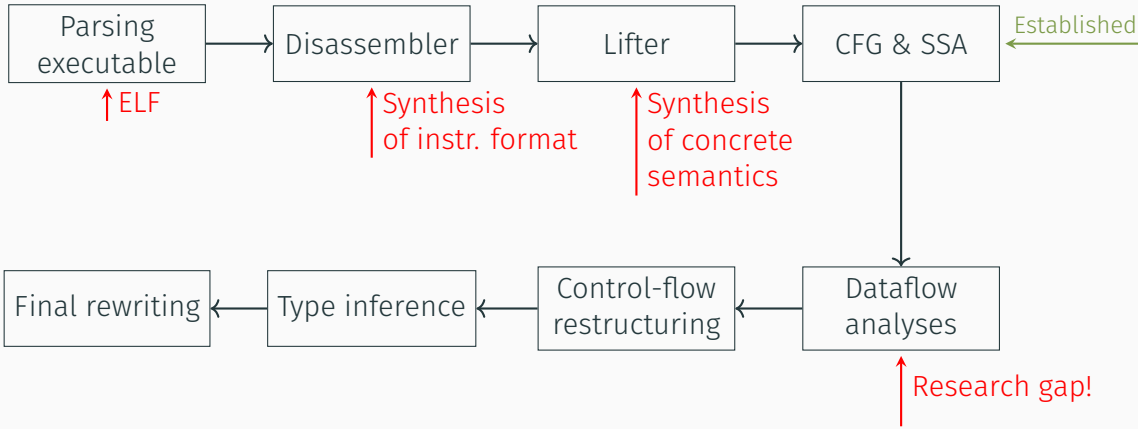
Further research



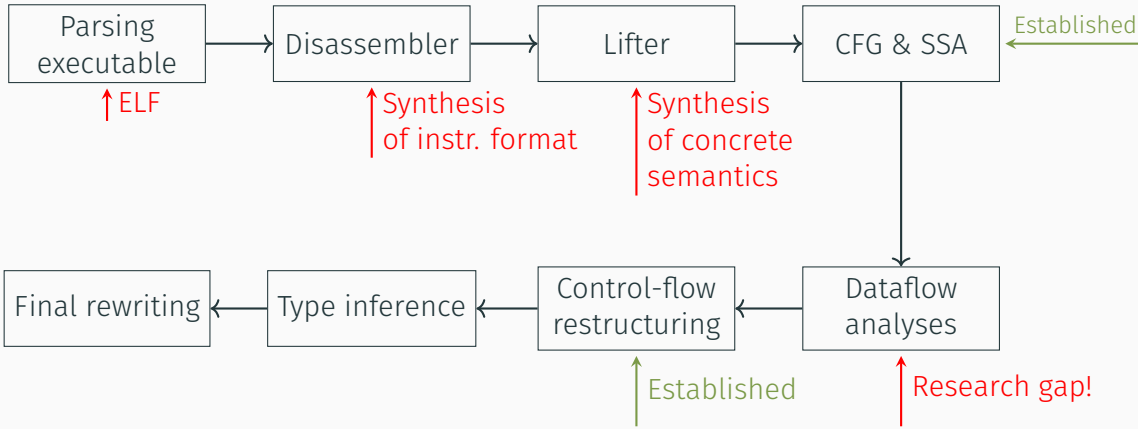
Further research



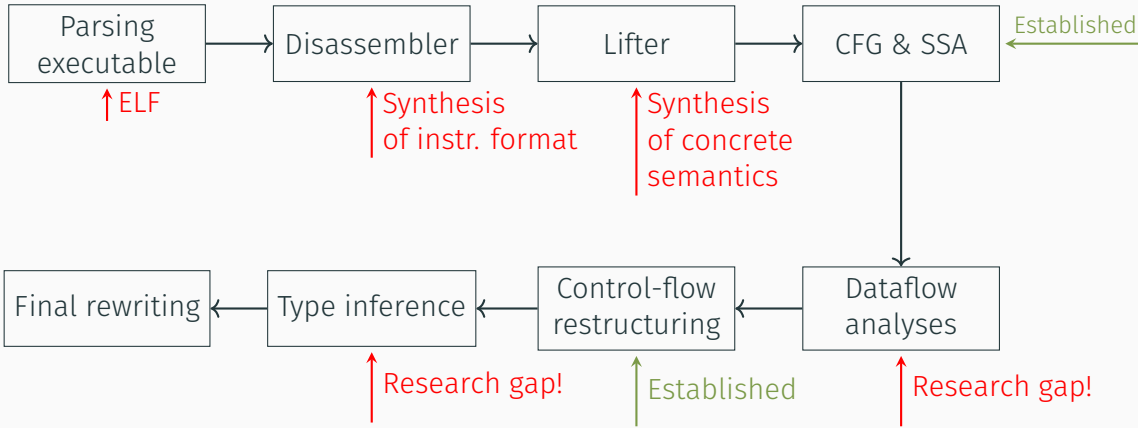
Further research



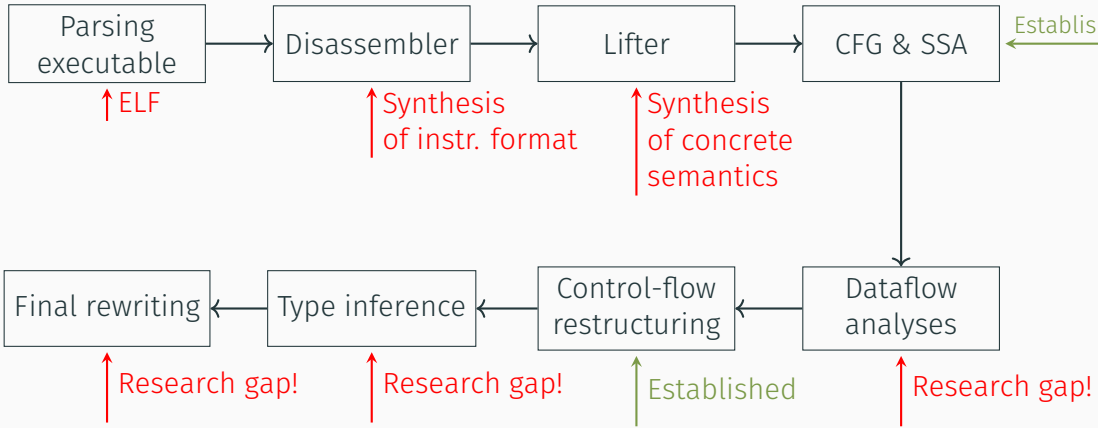
Further research



Further research



Further research



Summary

- introduced the CUDA architecture and its context

Summary

- introduced the CUDA architecture and its context
- PTX architecture under binja!

Summary

- introduced the CUDA architecture and its context
- PTX architecture under binja!
- SASS decompilation pipeline

Summary

- introduced the CUDA architecture and its context
- PTX architecture under binja!
- SASS decompilation pipeline

ptxninja

`https://github.com/seekbytes/ptxNinja`

Can't wait to see extension of this work!

Nicolò Altamura
@nicolodev
https://nicolo.dev