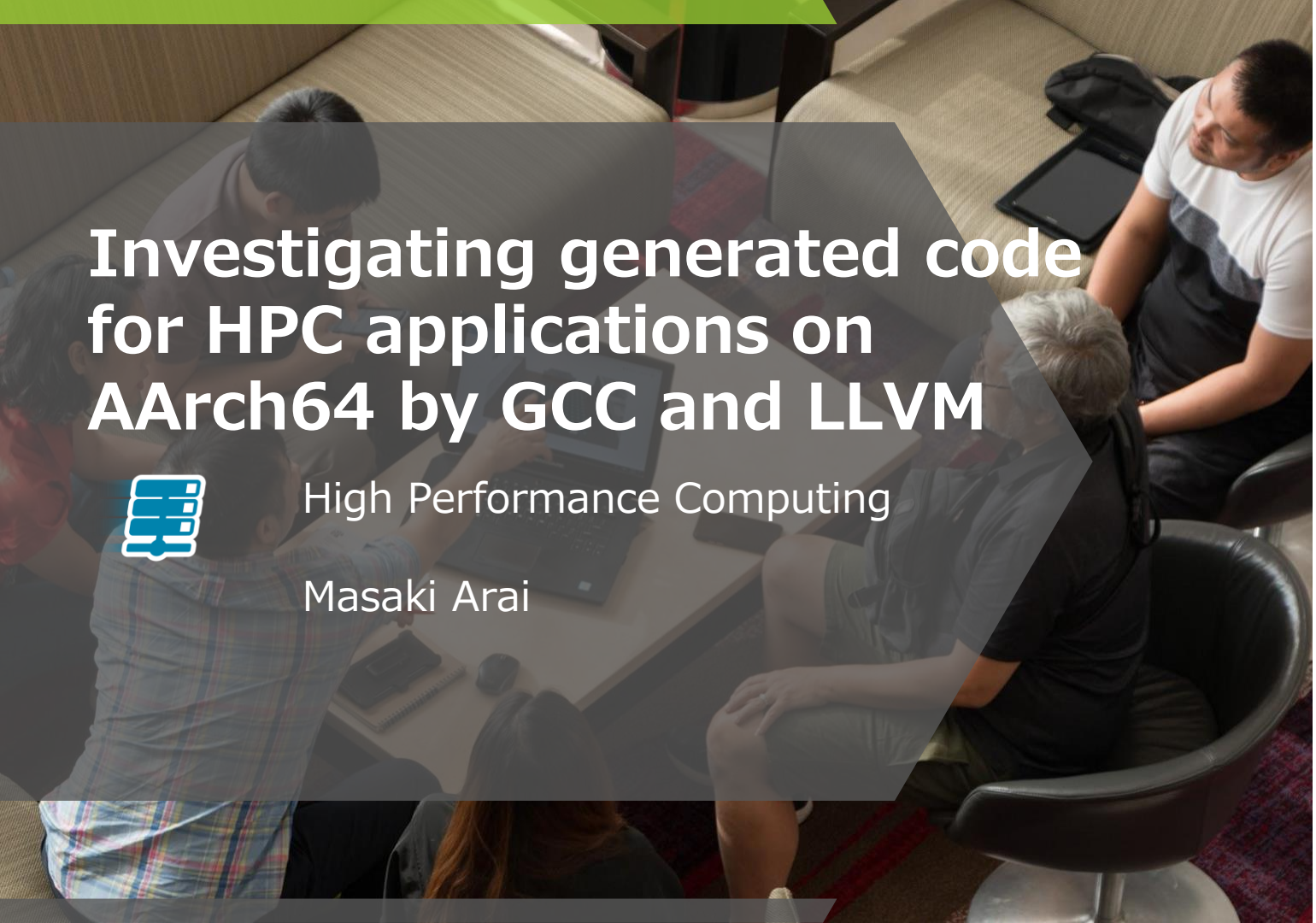


Investigating generated code for HPC applications on AArch64 by GCC and LLVM

High Performance Computing

Masaki Arai

Linaro connect
San Diego 2019

Outline

- Background
- Problems with LLVM compared to GCC
  - Redundant unconditional branches
  - Redundant basic blocks
  - Redundant conditional branches
  - Register selection
  - Software pipelining
- Summary

Background

- We are improving and enhancing LLVM for HPC applications.
- Many optimizations and patches are currently being introduced into LLVM for AArch64.
- For optimizations for HPC applications, GCC with a Fortran front end is superior to LLVM.
- We present the current problems of LLVM and propose solutions for them.
  - GCC version 9.2

    -O2 -fno-unroll-loops -march=armv8-a -mtune=thunderx2t99
  - Clang/LLVM version 9.0.0-rc3 with our extension

    -O2 -mllvm -unroll-count=0 -mcpu=thunderx2t99
- All sample code is extracted from HPC benchmark programs and we keep them on the following link:
  - https://github.com/Linaro/hcqc

Redundant Unconditional Branches

- There is no basic block to fall through into LBB0_2.
- Therefore, we can move the basic block with LBB0_2 and erase an unconditional branch.

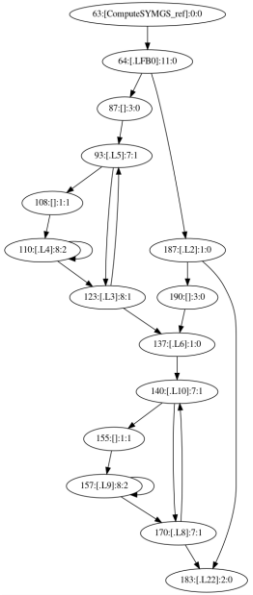
Current Code (LLVM)

```
            b           .LBB0_3
.LBB0_2:
            ldr         d2, [x11, x18]
            ...
            ...
            b.ne        .LBB0_5
            b           .LBB0_2
.LBB0_6:
```

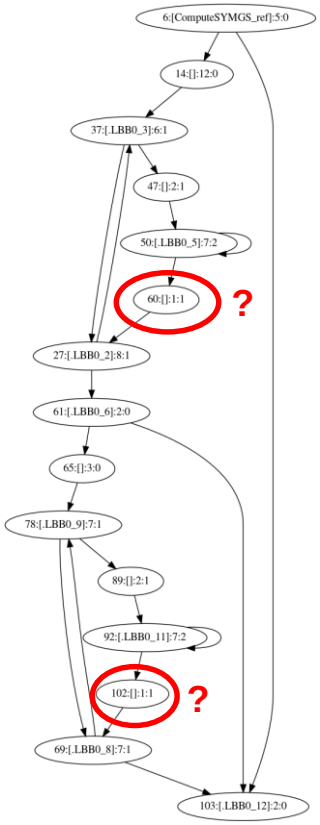
Improved Code by **BranchFolding** or **MachineBlockPlacement**

```
            b           .LBB0_3
...
...
            b.ne        .LBB0_5
.LBB0_2:
            ldr         d2, [x11, x18]
            ...
.LBB0_6:
```

GCC



LLVM

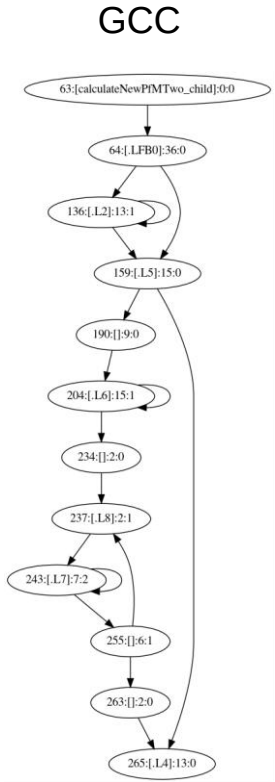


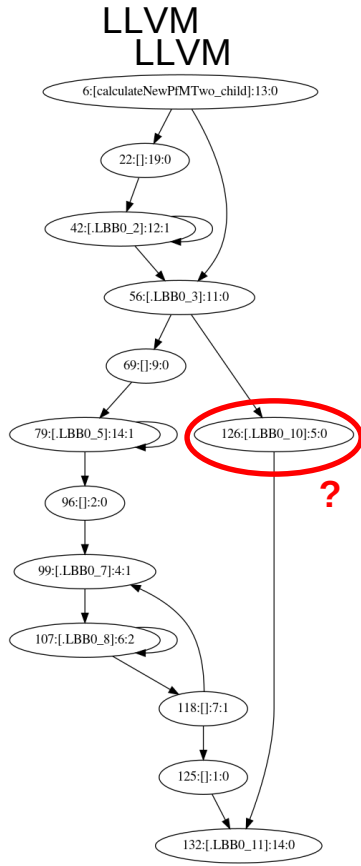
hpcg-3.0

Redundant Basic Blocks(1)

- Compared to GCC, there are additional basic blocks in the LLVM results.
- In this example, the additional basic block contains instructions that set some registers to zero(**xzr**).
- This is a problem in the process of converting SSA to non-SSA (**PhiElimination**).

```
.LBB0_10:
        fmov    d2, xzr
        fmov    d3, xzr
        fmov    d4, xzr
        fmov    d5, xzr
        fmov    d6, xzr
```





mVMC-mini-calculateNewPfMTwo_child

Redundant Basic Blocks(2)

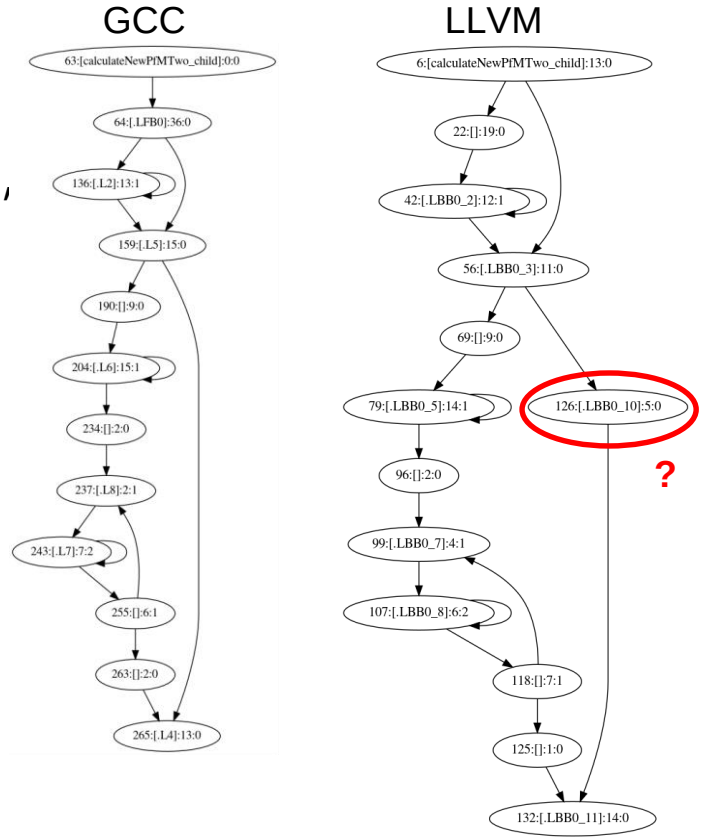
- **PhiElimination** inserts copy instructions for **PHI**, and src registers of copy instructions become zero registers(**xzr**).
- In such a case, these registers should be initialized in the upstream basic block.

```
p_a = p_b = q_a = q_b = bMa = 0.0;
....
for(msi=0;msi<nsize;msi++) {
  ....
  p_a += invM_ai * vec_ai;
  p_b += invM_bi * vec_ai;
  q_a += invM_ai * vec_bi;
  q_b += invM_bi * vec_bi;
}

for(msi=0;msi<nsize;msi++) {
  ...
}          PHI … <- (x1, x2)
ratio = invM_ab*vec_ba + invM_ab*bMa + p_a*q_b - p_b*q_a
```

copy x2 <- c

set x2 <- 0



mVMC-mini-calculateNewPfMTwo_child

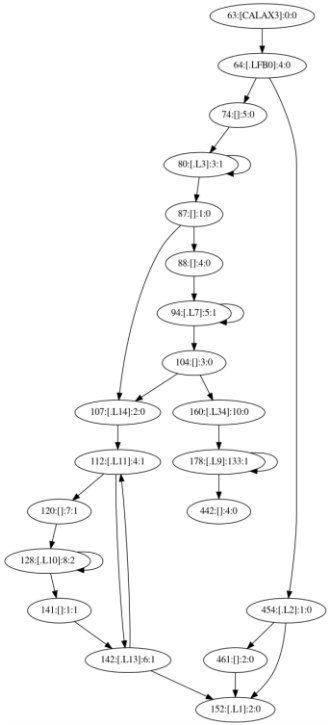
Redundant Conditional Branches(1)

- The CFG generated by LLVM is redundant compared to GCC.
- This is because the same comparison is performed many times.

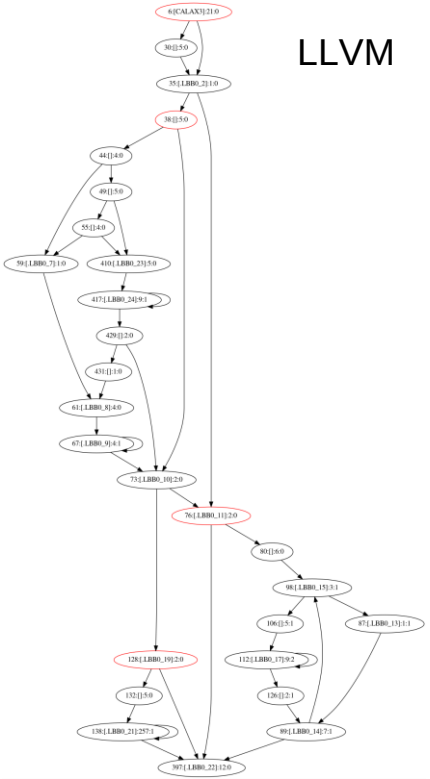
```
for (IP = 1; IP <= NP; IP++) {
   AS[IP]=0.0E0;
 }
 if (JUNROL == 0)
   goto L500;
 ....
 for (IP = 1; IP <= NP; IP++) {
   TS[IP]=S[IP];
 }
 if (NPPMAX > 30)
   goto L500;
 for (IP = 1; IP <= NP; IP++) {
    ...
 }
 goto L900;
L500:
 ...
 for (IP = 1; IP <= NP; IP++) {
   ...
 }
```

NP <= 0 ?

GCC



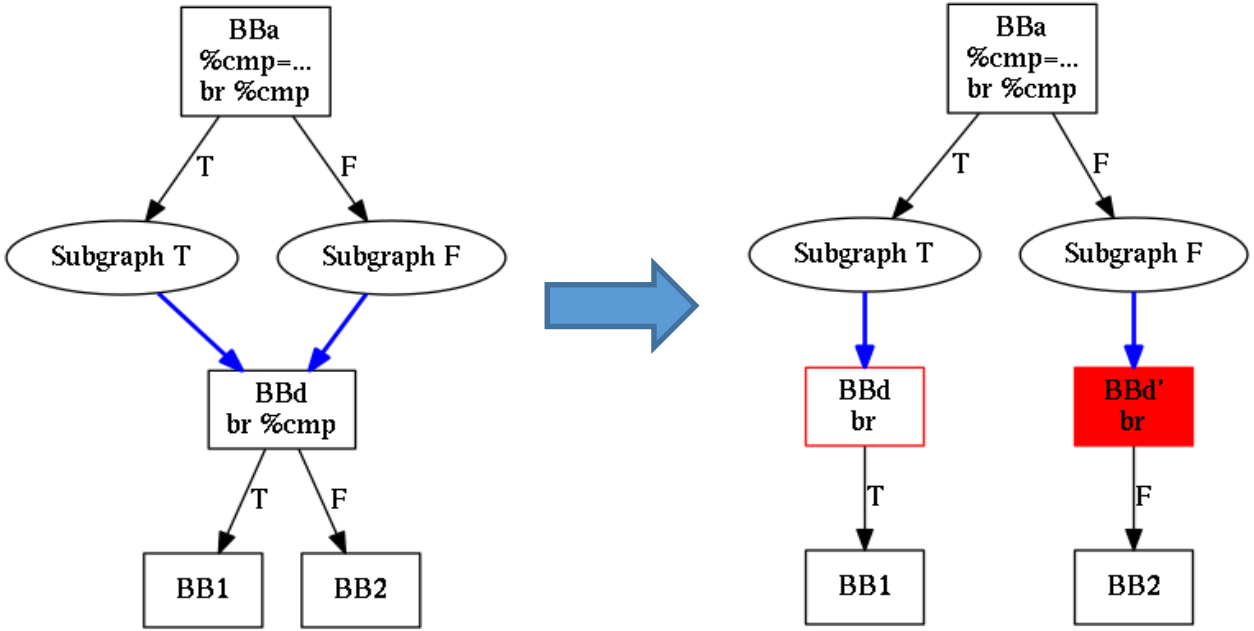
LLVM



ffb-mini-CALAX3

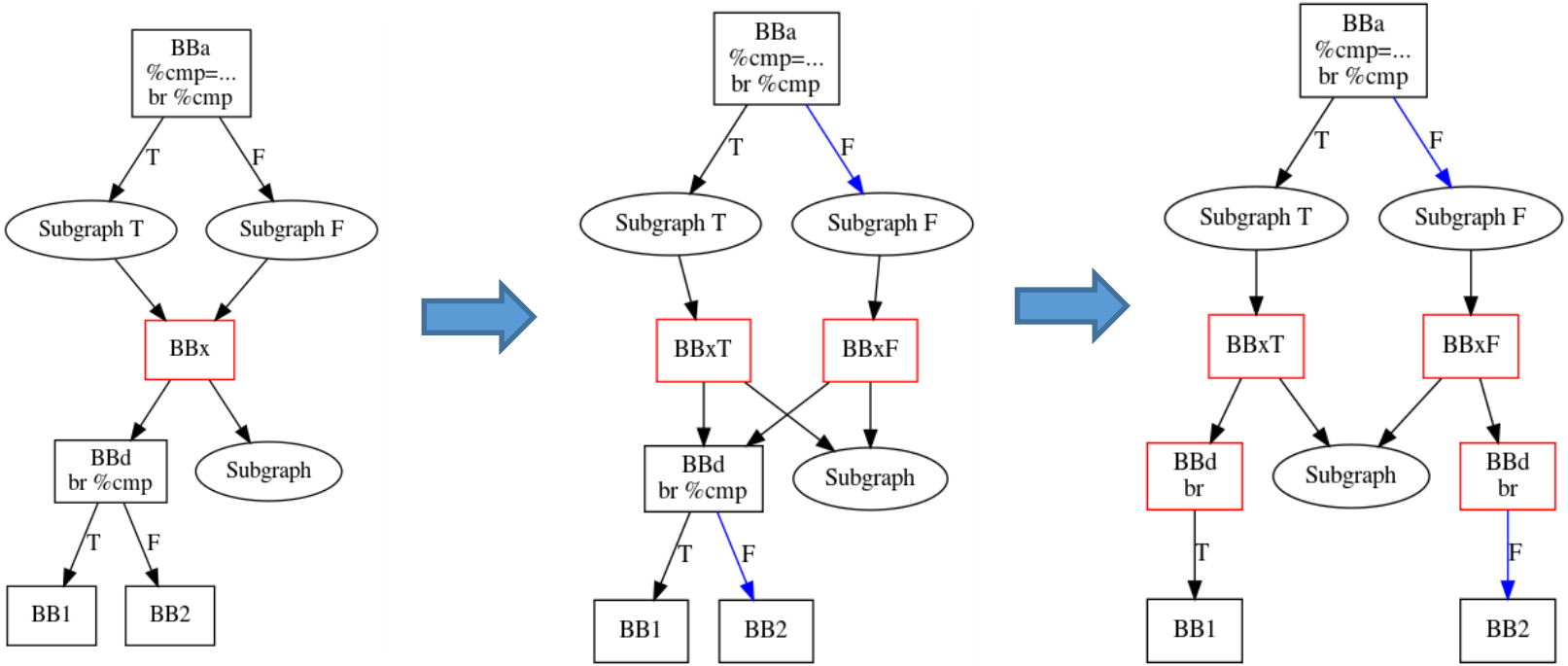
Redundant Conditional Branches(2)

- To address this problem, we implemented redundant branch eliminations using results of global value numbering.
- However, this was not enough.



Redundant Conditional Branches(3)

- For generating CFGs equivalent to GCC, we need the transformation shown in the figure.
- We are currently implementing additional optimizations to do this.



Register Selection(1)

- GCC uses floating point registers directly, but LLVM uses them indirectly.
- This is a problem with LLVM for handling fields of **struct**.

```
typedef struct { double re,im; } complex_dble;
typedef struct { double u[36]; } pauli_dble;

static complex_dble aa[36];

pauli_dble *m;
double sm,eps,*u;
int i,j,k;

   u=(*m).u;
   ...
   aa[6*i+i].re=*u;
   sm+=(*u)*(*u);
```

GCC

```
        ldr     d1, [x3]
        ...
        str     d1, [x0]
        ...
        add     x3, x3, 16
        ...
        ldr     d2, [x3, -8]
        fmul    d3, d2, d2
```

LLVM

```
        ldr     x14, [x0], #8
        fmov    d1, x14
       ...
        stp     x14, xzr, [x15]
        fmul    d1, d1, d1
```

QCDpbm-det_pauli_dble

Register Selection(2)

- A simple peephole optimization can solve this problem on either **IR** or **MIR**.
- The **IR** level optimization will increase optimization opportunities and reduce register waste.

LLVM IR

```
%3 = bitcast double* %u.0477 to i64*
%4 = load i64, i64* %3, align 8
...
%6 = bitcast %struct.complex_dble* %arrayidx to i64*
store i64 %4, i64* %6, align 8
...
%7 = bitcast i64 %4 to double
%mul6 = fmul double %7, %7
```

LLVM MIR

```
$x14 = LDRXpost $x0(tied-def 0), 8
$d1 = FMOVXDr $x14
...
STPXi $x14, $xzr, $x15, 0
$d1 = FMULDrr $d1, renamable $d1
```

QCDpbm-det_pauli_dble

Software Pipelining(1)

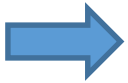
- Software pipelining is one of significant optimizations for HPC kernel loops.
- GCC has Swing Modulo Scheduling pass and some optimization path that facilitates it.
  - However, it seems that it has not been actively used or ported to various architectures.
- **MachinePipeliner** was introduced from LLVM 4.0.
  - It implements Swing Modulo Scheduling algorithm.
  - PowerPC developers currently propose many enhancement and improvement.
- We are porting **MachinePipeliner** to AArch64 by adding code specific to AArch64 for recognizing loop induction variables and conditional branches.
- This patch is not ready for LLVM upstream yet.
  - We need a target machine model for this optimization.
  - We are preparing our target machine model for **FUJITSU A64FX**(Armv8.2-A + SVE).

Software Pipelining(2)

- This is a simple result of **MachinePipeliner** for AArch64.
- We need much additional work, but we have already generated code that can run correctly.

```
define N 512

void
test (double *A, double *B, double c)
{
  int i = 0;
  do {
    A[i] = A[i] + c * B[i];
    i++;
  } while (i < N);
}
```

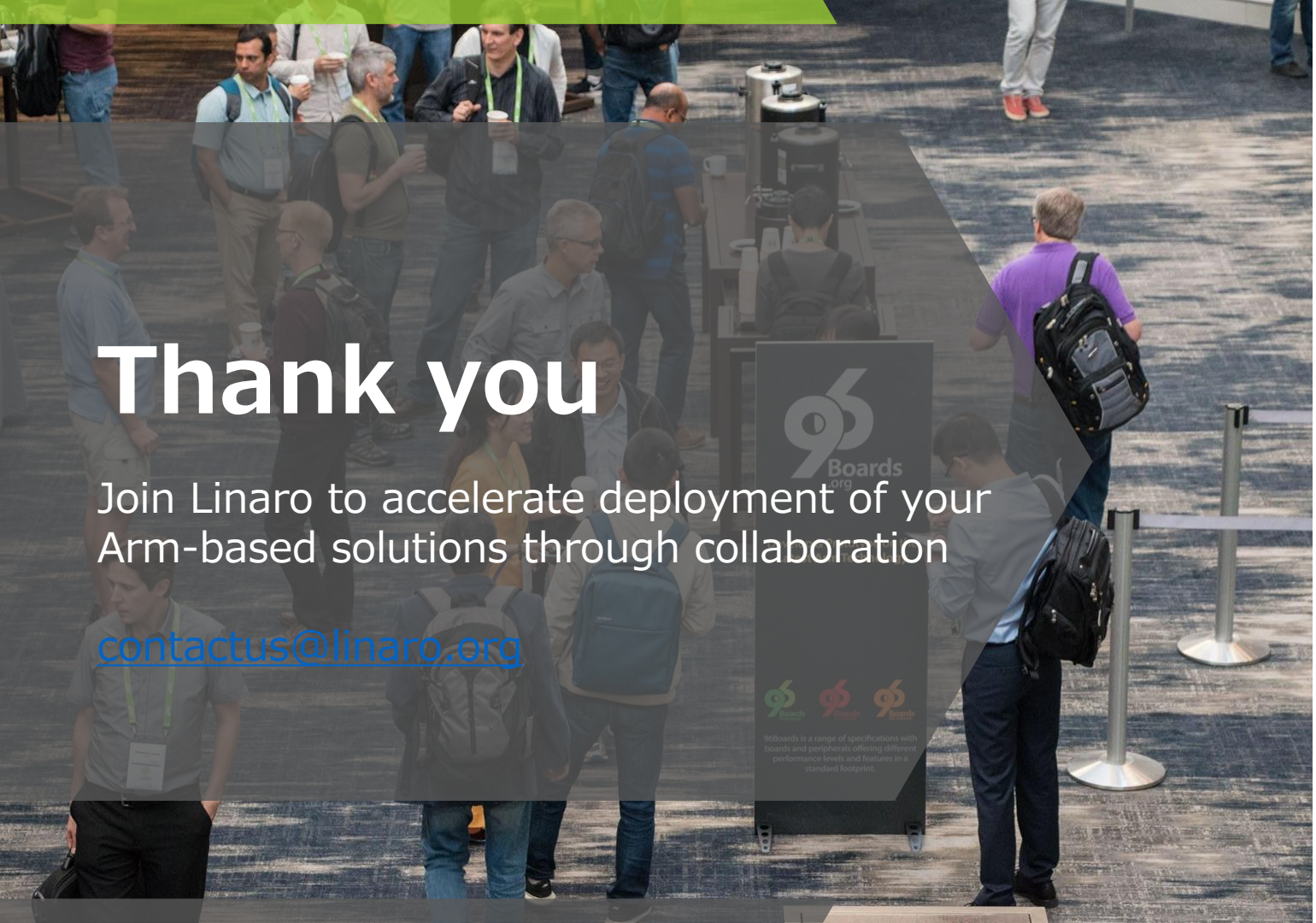


```
test:
        lsl     x8, xzr, #3
        mov     x9, xzr
        ldr     d2, [x1, x8]
        ldr     d1, [x0, x8]
        add     x9, x9, #1
        cmp     x9, #512
        fmul    d2, d2, d0
        fadd    d1, d1, d2
        .p2align 2
.LBB0_1:
        str     d1, [x0, x8]
        lsl     x8, x9, #3
        add     x9, x9, #1
        ldr     d1, [x1, x8]
        ldr     d2, [x0, x8]
        fmul    d1, d1, d0
        fadd    d1, d2, d1
        cmp     x9, #512
        b.ne    .LBB0_1
        str     d1, [x0, x8]
        ret
```

II=8

Summary

- LLVM has reached a sufficient level as a compiler for system programming.
- However, there are several problems for HPC applications.
- For optimizations for HPC applications, GCC with a Fortran front end is superior to LLVM.
- The problems we point out are problems with a small number of instructions.
- However, these problems occur around loops, which are hot spots for HPC applications.
- We will propose solutions for these problems to LLVM's upstreaming.



Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org

Linaro connect
San Diego 2019