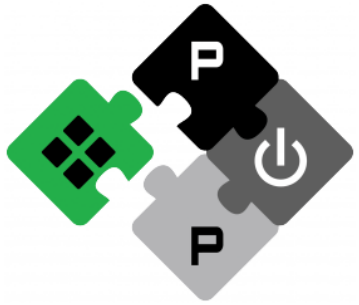


PULP PLATFORM

Open Source Hardware, the way it should be!

Mastering the PULP GCC toolchain

Introduction to the compilation toolchan and Performance-driven optimization techniques

Giuseppe Tagliavini <giuseppe.tagliavini@unibo.it>

 http://pulp-platform.org  @pulp_platform

ETH zürich

Outline

- **Introduction to the PULP GCC toolchain**
- Downloading and building the toolchain
- RISC-V and PULP compiler options
- Performance-driven optimization techniques
- Understanding the compiler optimization passes
- Common issues and best practices

The PULP GCC toolchain

- Available on GitHub:
  https://github.com/pulp-platform/pulp-riscv-gnu-toolchain

- The toolchain includes these components:
  - **GCC 7.1.1**
  - **Binutils 2.28**
  - **Newlib 2.5.0**
  - Glibc 2.26
  - DejaGNU 1.5.3

- Integration with the PULP SDK by means of an environment variable
  - Set the PULP_RISCV_GCC_TOOLCHAIN variable to the install folder (i.e., the parent of the bin folder)

Outline

- Introduction to the PULP GCC toolchain
- **Downloading and building the toolchain**
- RISC-V and PULP compiler options
- Performance-driven optimization techniques
- Understanding the compiler optimization passes
- Common issues and best practices

Downloading and building the toolchain

- Instructions are available at: https://github.com/pulp-platform/pulp-riscv-gnu-toolchain
- Steps to build the toolchain:

1. Install required packages for Ubuntu or Centos
2. Clone the repository and all submodules:
   git clone --recursive https://github.com/pulp-platform/pulp-riscv-gnu-toolchain
3. Configure:
   ./configure --prefix=<INSTALL_DIR> --with-arch=rv32imc --with-cmodel=medlow --enable-multilib
4. Add the bin directory to the path variable:
   export PATH=<INSTALL_DIR>/bin:$PATH
5. Build the toolchain:
   make

Libraries

- GCC low-level runtime library (**libgcc**)
  - Handle arithmetic operations that the target processor cannot perform directly (e.g., floating-point emulation)
  - Location:
    <INSTALL_DIR>/lib/gcc/riscv32-unknown-elf/7.1.1/rv32imfcxpulpv2/ilp32
- Newlib is a C standard library implementation intended for use on embedded systems
  - Includes several components (**libc**, **libm**, …)
  - Location:
    <INSTALL_DIR>/riscv32-unknown-elf/lib/rv32imfcxpulpv2/ilp32
- The building two described in the previous slide generates several variants of these libraries, corresponding to different architectures and ABIs

Outline

- Introduction to the PULP GCC toolchain
- Downloading and building the toolchain
- **RISC-V and PULP compiler options**
- Performance-driven optimization techniques
- Understanding the compiler optimization passes
- Common issues and best practices

RISC-V compiler options

- **-march=ISA-string**
  Generate code for given RISC-V ISA
- **-mabi=ABI-string**
  Specify integer and floating point calling convention
- **-mtune=processor-string**
  Optimize the output for the given microarchitecture name
- **-mcmodel=medlow**
  Generate code for the medium-low code model (default). The program and its statically defined symbols must lie within a single 2 GiB address range and must lie between absolute addresses -2 GiB and +2 GiB. Programs can be statically or dynamically linked
- **-mcmodel=medany**
  Generate code for the medium-any code model. The program and its statically defined symbols must be within any single 2 GiB address range.
  Programs can be statically or dynamically linked

PULP compiler options (1/2)

- **-mPE=num**
  Set the number of Pes in the PULP cluster
- **-mFC=0/1**
  0: without FC, 1: with FC
- **-mL2=size**
  Set L2 size
- **-mL1Cl=size**
  Set cluster L1 size
- **-mnopostmod**
  Disable post modification support for pointer arithmetic
- **-mnoindregreg**
  Disable load/store with register offset for pointer arithmetic
- **-mnovect**
  Disable the support to packed-SIMD instructions

PULP compiler options (2/2)

- **-mnohwloop**
  Disable the hardware loop support
- **-mhwloopmin=num**
  Minimum numbwe of instructiona in hardware loops (default is 2)
- **-mhwloopalign**
  Force memory alignment of hardware loops

- Other options to disable specific instructions:
  -mnomac, -mnopartmac, -mnominmax, -mnoabs, -mnobitop, -mnosext, -mnoclip, -mnoaddsubnormround, -mnomulmacnormround, -mnoshufflepack

Outline

- Introduction to the PULP GCC toolchain
- Downloading and building the toolchain
- RISC-V and PULP compiler options
- **Performance-driven optimization techniques**
- Understanding the compiler optimization passes
- Common issues and best practices

Case study: Matrix multiplication v1.0

```
void matmul(int *__restrict__ A, int *__restrict__ B, int *__restrict__ C) {
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      C[i*N+j] = 0;
      for (int k = 0; k < K; k++) {
         C[i*N+j] += A[i*K+k] * B[k*N+j];
      } //k
    }//j
  }//i
}
```

Elementary operation → multiply-and-accumulate (MAC)

Space complexity → $\mathcal{O}(M \cdot N)$

Time complexity → $\mathcal{O}(M \cdot N \cdot K)$

Deriving an ideal model

- How many *machine operations* are required to complete an elementary operation?
  - To answer this question, we need to define a *machine model*
  - Our reference machine: a **RI5CY core**
- Different ideal models:
  - 1 elementary operation == 2 lw + 1 MAC + 1 sw → M*N*K *4
  - 1 elementary operation == 2 lw + 1 MAC + (1/K) sw → M*N*K *3 + N*M
- **To derive the best model, we need a good understanding at assembly level**

Assembly v1.0 (-march=rv32imcXpulpv2)

- **Kernel parameters of v1.0 are NOT function parameters**

#define M 50

#define N 50

#define K 30

```
...
1c0086ea:       0168407b        lp.setup    x0,a6,1c008716
1c0086ee:       41ce87b3        sub         a5,t4,t3
1c0086f2:       17f1            addi        a5,a5,-4
1c0086f4:       8389            srli        a5,a5,0x2
1c0086f6:       0008a22b        p.sw        zero,4(a7!)
1c0086fa:       861a            mv          a2,t1
1c0086fc:       86f2            mv          a3,t3
1c0086fe:       4701            li          a4,0
1c008700:       0785            addi        a5,a5,1
1c008702:       0067c0fb        lp.setup    x1,a5,1c00870e
1c008706:       0046a50b        p.lw        a0,4(a3!)
1c00870a:       0c86258b        p.lw        a1,200(a2!)
1c00870e:       42b50733        p.mac       a4,a0,a1
1c008712:       fee8ae23        sw          a4,-4(a7)
1c008716:       0311            addi        t1,t1,4
...
```

hardware loops (id, iterations, end) to remove branch overhead

address pre/post-increment to optimize memory access with regular patterns

Performance Counters

- **Performance counters** a set of special-purpose registers built into a core to count hardware-related events with *high precision* and *low overhead*
  - **Execution cycles**
  - **Instructions**
  - Active cycles
  - External loads
  - TCDM contentions
  - **Load stalls**
  - **I-cache misses**
  - FPU contentions/dependencies/write-back stalls

Using the performance counters

```
INIT_STATS();
```
1. Declare variables

```
for(int i=0; i<M*K; i++) A[i] = 1;
for(int i=0; i<K*N; i++) B[i] = 1;
```

```
BEGIN_STATS_LOOP();
```
2. Repeat measures in a loop

2b. Add code to re-init data at each measuring iteration (if needed!)

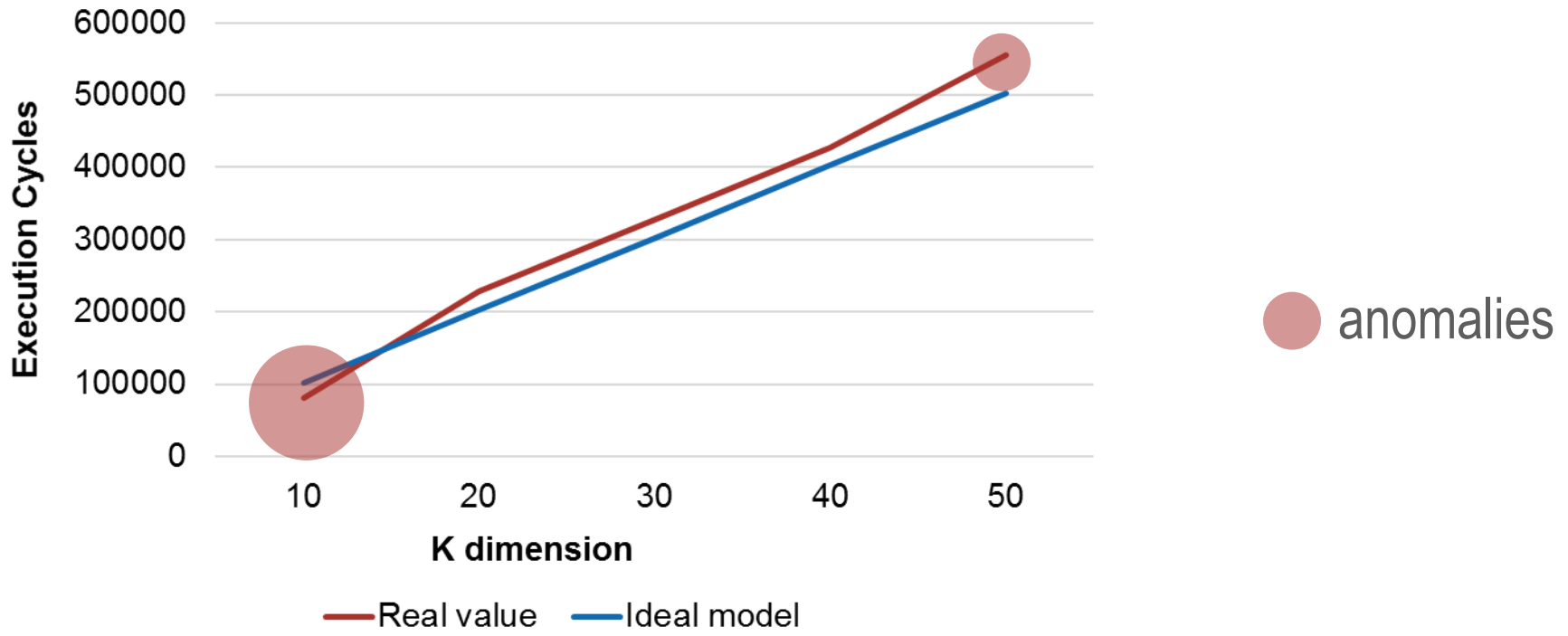
```
START_STATS();
```
3. Start measuring

```
matmul(A, B, C, M, N, K);
STOP_STATS();
```
4. Pause measuring (accumulate into variables)

```
END_STATS_LOOP();
```
5. End of measuring loop

Real values VS ideal model

- Algorithm setup: M=50, N=50, K=10..50
- Target platform: PULP-Open on FPGA target



Anomaly #1

- Comparing K=10 to K=20
  - The number of instructions executed when K=10 is less than half. This is good, but why?
  - And what about the I-cache misses?

V1.0, M=50, N=50, K=10

[0] cycles = 80778
[0] instr = 53326
[0] active cycles = 80778
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 25000
[0] imiss = 2449

V1.0, M=50, N=50, K=20

[0] cycles = 227970
[0] instr = 177869
[0] active cycles = 227970
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 50000
[0] imiss = 0

Assembly (v1.0 M=50 N=50 K=10)

```
...
1c0080f6:       02a6c0fb        lp.setup   x1,a3,1c00814a
1c0080fa:       0047278b        p.lw    a5,4(a4!)
1c0080fe:       02f287b3        mul     a5,t0,a5
1c008102:       0c472983        lw      s3,196(a4)
1c008106:       433f87b3        p.mac   a5,t6,s3
1c00810a:       18c72983        lw      s3,396(a4)
1c00810e:       433f07b3        p.mac   a5,t5,s3
1c008112:       25472983        lw      s3,596(a4)
1c008116:       433e87b3        p.mac   a5,t4,s3
1c00811a:       31c72983        lw      s3,796(a4)
1c00811e:       433e07b3        p.mac   a5,t3,s3
1c008122:       3e472983        lw      s3,996(a4)
1c008126:       433307b3        p.mac   a5,t1,s3
1c00812a:       4ac72983        lw      s3,1196(a4)
1c00812e:       433887b3        p.mac   a5,a7,s3
1c008132:       57472983        lw      s3,1396(a4)
1c008136:       433807b3        p.mac   a5,a6,s3
1c00813a:       63c72983        lw      s3,1596(a4)
1c00813e:       433507b3        p.mac   a5,a0,s3
1c008142:       70472983        lw      s3,1796(a4)
1c008146:       433587b3        p.mac   a5,a1,s3
1c00814a:       00f6222b        p.sw    a5,4(a2!)
...
```

The loop start is not aligned → We have a cache penalty in the RI5CY core

The inner loop is totally unrolled → There is no loop overhead, we save 50*50*9=22500 instructions!

Removing anomaly #1

- In this case, we can remove the I$ misses using the -mhwloopalign flag

V1.0, M=50, N=50, K=10

[0] cycles = 80778
[0] instr = 53326
[0] active cycles = 80778
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 25000
[0] imiss = 2449

V1.0, M=50, N=50, K=10 -mhwloopalign

[0] cycles = 78330
[0] instr = 53327
[0] active cycles = 78330
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 25000
[0] imiss = 0

Anomaly #2

- **Comparing K=40 to K=50**
  - The number of cycles executed when K=50 is higher than expected
  - I-cache misses are very high

V1.0, M=50, N=50, K=40

[0] cycles = 427970
[0] instr = 327869
[0] active cycles = 427970
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 100000
[0] imiss = 0

V1.0, M=50, N=50, K=50

[0] cycles = 554784
[0] instr = 403024
[0] active cycles = 554784
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 125000
[0] **imiss = 127114**

Removing anomaly #2

- Again, we can remove the I$ misses using the -mhwloopalign flag

V1.0, M=50, N=50, K=50

[0] cycles = 554784
[0] instr = 403024
[0] active cycles = 554784
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 125000
[0] **imiss = 127114**

V1.0, M=50, N=50, K=50 -mhwloopalign

[0] cycles = 543175
[0] instr = 403074
[0] active cycles = 543175
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 125000
[0] imiss = 7300

Removing stalls

- **Load stalls** are the major source of overhead in the version 1.0 of the kernel
- These stalls are due to the latency of memory accesses → a RI5CY core requires 1 additional cycle to access its local memory

```
lp.setup        x1,a5,1c0080ec
p.lw    a0,4(a3!)
p.lw    a1,200(a2!)
p.mac   a4,a0,a1
```

- Solution: apply manual **loop unrolling**

Matrix multiplication v2.0

```
void matmul(int *__restrict__ A, int *__restrict__ B, int *__restrict__ C) {
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      C[i*N+j] = 0;
      for (int k = 0; k < K; k+=2) {
        int A1 = A[i*K+k], A2 = A[i*K+k+1];
        int B1 = B[k*N+j], B2 = B[(k+1)*N+j];
        asm volatile(""::::"memory");
        C[i*N+j] += A1 * B1;
        C[i*N+j] += A2 * B2;
      } //k
    }//j
  }//i
}
```

Memory barrier →
Inhibits compiler reordering of memory accesses

Measuring performance of v2.0

- We have not removed all the stalls. Why?

V2.0, M=50, N=50, K=10

[0] cycles = 78329
[0] instr = 53326
[0] active cycles = 78329
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 25000
[0] imiss = 0

V2.0, M=50, N=50, K=20

[0] cycles = 266789
[0] instr = 194139
[0] active cycles = 266789
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 72500
[0] imiss = 49

Assembly v2.0 (M=50 N=50 K=50)

- Loop unrolling with the addition of a memory barrier introduced an unexpected behavior

```
lp.setup        x1,a4,1c008748 <matmul.constprop.0+0xa8>
1c00872c:       00862a8b            p.lw    s5,8(a2!)
1c008730:       0085230b            p.lw    t1,8(a0!)
1c008734:       1905aa0b            p.lw    s4,400(a1!)
1c008738:       1908288b            p.lw    a7,400(a6!)
1c00873c:       ffc6a783            lw      a5,-4(a3)
1c008740:       434a87b3            p.mac   a5,s5,s4
1c008744:       431307b3            p.mac   a5,t1,a7
1c008748:       fef6ae23            sw      a5,-4(a3)
```

Load stall

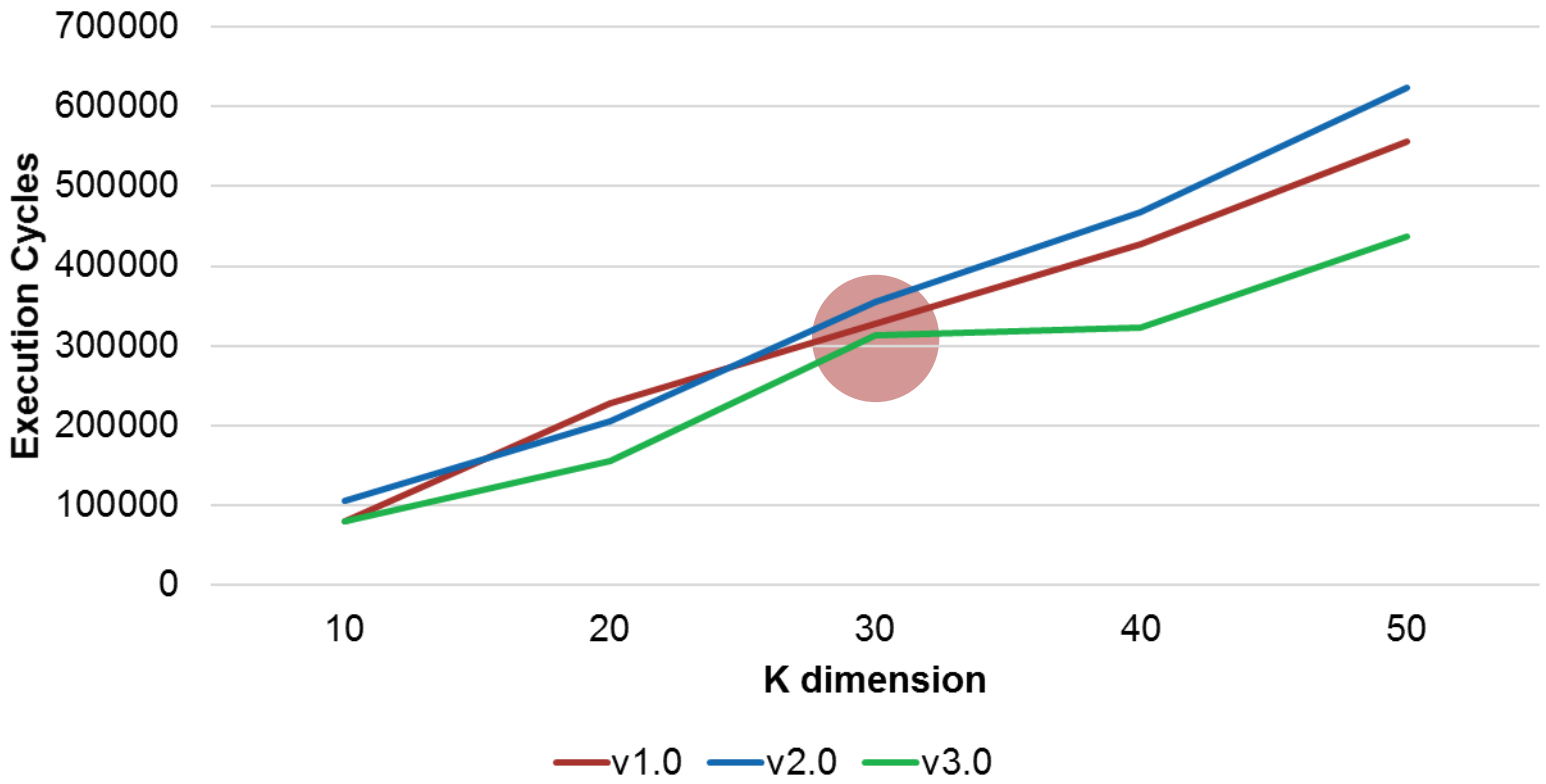
Writing back to memory
In the inner loop

Matrix multiplication v3.0

```
void matmul(int *__restrict__ A, int *__restrict__ B, int *__restrict__ C) {
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      int val = 0;
      for (int k = 0; k < K; k+=2) {
         int A1 = A[i*K+k], A2 = A[i*K+k+1];
         int B1 = B[k*N+j], B2 = B[(k+1)*N+j];
         asm volatile("":::"memory");
         val += A1 * B1;
         val += A2 * B2;
      } //k
      C[i*N+j] = val;
    }//j
  }//i
}
```

Experimental results

- There is an anomaly at K=30, but we are not able to remove it → the compiler is performing some kind of unrolling…



Outline

- Introduction to the PULP GCC toolchain
- Downloading and building the toolchain
- RISC-V and PULP compiler options
- Performance-driven optimization techniques
- **Understanding the compiler optimization passes**
- Common issues and best practices

Optimization passes

- **Optimization pass:** an algorithm that transforms a code to produce a **semantically equivalent** code that uses fewer resources and/or executes faster
- Each optimization level is an **ordered list** of optimization passes
- To get the list of applied optimization passes:
  - **riscv32-unknown-elf-gcc -Q -O2 -v --help=optimizers**
  - Add **-fdump-passes** to the compiler parameters (or to CFLAGS Makefile variable)
- Adding/removing optimization passes:
  - To add an optimization pass: -f*passname*
  - To remove an optimization pass: -fno-*passname*
  - Reference: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

From O3 to O2 (and back)

- Suppose to remove from an O3 compilation line all the optimization passes added w.r.t. O2. Are we executing using O2 optimization level?
- The answer is… NO!!!
- **Some optimization options that depend on the optimization level are not tunable with command line parameters**

```
/* There is no assumptions if the loop is known to be finite.  */
if (!integer_zerop (niter->assumptions)
    && loop_constraint_set_p (loop, LOOP_C_FINITE))
  niter->assumptions = boolean_true_node;

if (optimize >= 3)
  {
    niter->assumptions = simplify_using_outer_evolutions (loop,
                                                          niter->assumptions);
    niter->may_be_zero = simplify_using_outer_evolutions (loop,
                                                          niter->may_be_zero);
    niter->niter = simplify_using_outer_evolutions (loop, niter->niter);
  }

niter->assumptions
       = simplify_using_initial_conditions (loop,
                                            niter->assumptions);
niter->may_be_zero
       = simplify_using_initial_conditions (loop,
                                            niter->may_be_zero);
```

Useful flags (1/2)

- **-mno-memcpy**
  Disable the automatic use of memcpy and allows the compiler to inline constant-sized copies
- **-fno-tree-vectorize**
  Disables code transformations for automatic vectorization
- **-fno-tree-loop-distribution -fno-tree-loop-distribute-patterns**
  Disable the splitting of the loop workload into multiple adjacent loops (preliminary step for automatic vectorization and parallelization)

Useful flags (2/2)

- **-fno-tree-ch**
  Disables loop header copying on trees
- **-fno-tree-loop-im**
  Disables loop invariant motion on trees of complex instructions
- **-fno-unswitch-loops**
  Avoids to move branches with loop invariant conditions out of the loop, with duplicates of the loop on both branches (modified according to result of the condition)

Applying compiler optimization at fine grain

- In GCC we can restrict optimization parameters to single functions using this syntax:

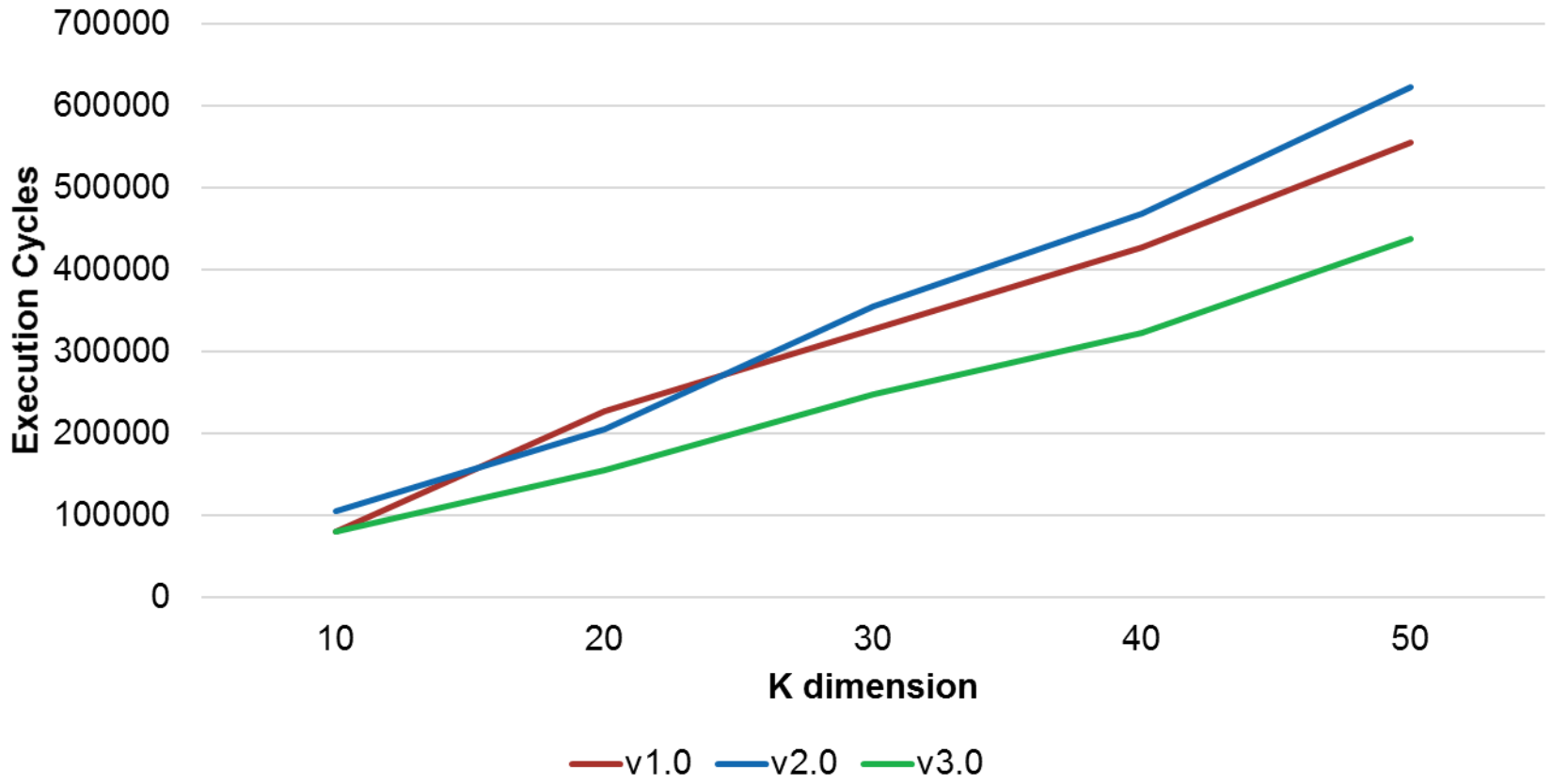
```
#pragma GCC push_options
#pragma GCC optimize ("-O2")

void kernel(...){
    ...
}

#pragma GCC pop_options
```

- This technique can be used to change the optimization level but also to enable/disable specific passes using flags (see previous slides)

A solution to the anomaly in code v3.0

- We found an anomaly in the code for K=30
- We can try to apply the optimizations flags in the previous slides
- Final solution: **we can force O2 optimization level for the kernel since this anomaly is related to a hidden parameter that is not tunable at command line**

Experimental results v3.0 (fixed)



Outline

- Introduction to the PULP GCC toolchain
- Downloading and building the toolchain
- RISC-V and PULP compiler options
- Performance-driven optimization techniques
- Understanding the compiler optimization passes
- **Common issues and best practices**

Issue #1: Moving from constants to parameters

- To make our code really parametric, we will probably change the function signature as follows:

```
void matmul(int *__restrict__ A, int *__restrict__ B, int *__restrict__ C,
            int M1, int N1, int K1);
```

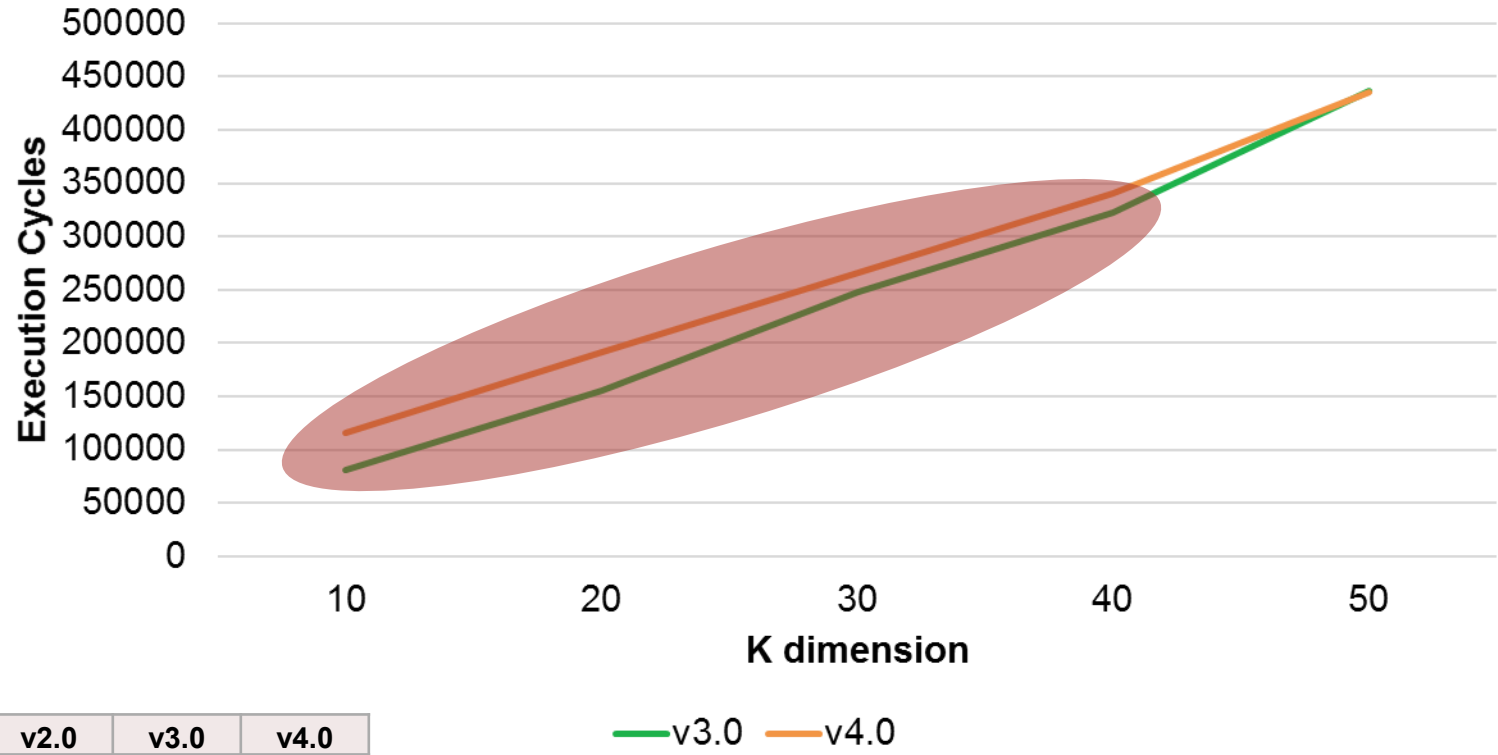
- However, if we test this function with constant parameters, the compiler will apply **constant propagation** deriving a version of the function equivalent to the previous one
- To avoid this problem, we can invoke the function passing **volatile** variables:

```
volatile int m = M;
volatile int n = N;
volatile int k = K;
matmul(A, B, C, m, n, k);
```


Matrix multiplication v4.0

```
void matmul(int *__restrict__ A, int *__restrict__ B, int *__restrict__ C,
            int M1, int N1, int K1) {

  for (int i = 0; i < M1; i++) {

    for (int j = 0; j < N1; j++) {

      int val = 0;

      for (int k = 0; k < K1; k+=2) {

         int A1 = A[i*K1+k], A2 = A[i*K1+k+1];

         int B1 = B[k*N1+j], B2 = B[(k+1)*N1+j];

         asm volatile(""::::"memory");

         val += A1 * B1;

         val += A2 * B2;

      } //k

      C[i*N1+j] = val;

    }//j
  }//i
}
```


Experimental results v4.0



K	v1.0	v2.0	v3.0	v4.0
10	80778	105318	80322	115849
20	227970	205347	155351	190851
30	327970	355535	248035	265851
40	427970	468035	323035	340851
50	555812	623109	437120	435922

Performance counters v4.0

- Looking at the performance counters, there is no clear anomaly…

V4.0, M=50, N=50, K=10

[0] cycles = 115849
[0] instr = 115696
[0] active cycles = 115849
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 0
[0] imiss = 49

V4.0, M=50, N=50, K=20

[0] cycles = 190851
[0] instr = 190698
[0] active cycles = 190851
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 0
[0] imiss = 49

V4.0, M=50, N=50, K=30

[0] cycles = 265851
[0] instr = 265698
[0] active cycles = 265851
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 0
[0] imiss = 49

Assembly v4.0

```
...
1c008100:       024fc07b            lp.setup        x0,t6,1c008148
1c008104:       4781                li      a5,0
1c008106:       02c05f63            blez    a2,1c008144
1c00810a:       fff60713            addi    a4,a2,-1
1c00810e:       8305                srli    a4,a4,0x1
1c008110:       00160313            addi    t1,a2,1
1c008114:       4e09                li      t3,2
1c008116:       00590833            add     a6,s2,t0
1c00811a:       85a2                mv      a1,s0
1c00811c:       8896                mv      a7,t0
1c00811e:       86a6                mv      a3,s1
1c008120:       4781                li      a5,0
1c008122:       0705                addi    a4,a4,1
1c008124:       05c34563            blt     t1,t3,1c00816e
1c008128:       00c740fb            lp.setup        x1,a4,1c008140
1c00812c:       0086af0b            p.lw    t5,8(a3!)
1c008130:       0085ae0b            p.lw    t3,8(a1!)
1c008134:       2188fe8b            p.lw    t4,s8(a7!)
1c008138:       2188730b            p.lw    t1,s8(a6!)
1c00813c:       43df07b3            p.mac   a5,t5,t4
1c008140:       426e07b3            p.mac   a5,t3,t1
1c008144:       00f3a22b            p.sw    a5,4(t2!)
...
```

12 instructions (+4)
Branches!!!???

How to fix v4.0

- The compiler must be sure that a cycle is executed at least once
  - This is simple with constant loop bounds…
  - …But it is not obvious with variables!

- Solution: use a **do…while** construct
  - This enforces the «at least once» execution semantic

Matrix multiplication v5.0

```
void matmul(int *__restrict__ A, int *__restrict__ B, int *__restrict__ C,
            int M1, int N1, int K1) {
  int i = 0;
  do
  {
    int j = 0;
    do
    {
      int val = 0;
      int k = 0;
      do
      {
         int A1 = A[i*K1+k], A2 = A[i*K1+k+1];
         int B1 = B[k*N1+j], B2 = B[(k+1)*N1+j];
         asm volatile(""::::"memory");
         val += A1 * B1;
         val += A2 * B2;
         k += 2;
      } while(k < K1);
      C[i*N1+j] = val;
      j++;
    } while(j < N1);
    i++;
  } while (i < M1);
}
```

Assembly v5.0

```
...
1c0080ec:       0223c07b            lp.setup        x0,t2,1c008130
1c0080f0:       fff60713            addi    a4,a2,-1
1c0080f4:       8305                srli    a4,a4,0x1
1c0080f6:       00160e13            addi    t3,a2,1
1c0080fa:       4e89                li      t4,2
1c0080fc:       00598833            add     a6,s3,t0
1c008100:       86a6                mv      a3,s1
1c008102:       8896                mv      a7,t0
1c008104:       834a                mv      t1,s2
1c008106:       4781                li      a5,0
1c008108:       0705                addi    a4,a4,1
1c00810a:       05de4563            blt     t3,t4,1c008154 <matmul.constprop.0+0xb4>
1c00810e:       0001                nop
1c008110:       00c740fb            lp.setup        x1,a4,1c008128
1c008114:       00832f8b            p.lw    t6,8(t1!)
1c008118:       0086ae8b            p.lw    t4,8(a3!)
1c00811c:       2188ff0b            p.lw    t5,s8(a7!)
1c008120:       21887e0b            p.lw    t3,s8(a6!)
1c008124:       43ef87b3            p.mac   a5,t6,t5
1c008128:       43ce87b3            p.mac   a5,t4,t3
1c00812c:       00f4222b            p.sw    a5,4(s0!)
...
```

-1 branch + 1 nop!!!

Fixes to v5.0

- Whenever it is possible, set loop steps to 1

- Try to move algebraic computations to outer loops (i.e., array indexes)

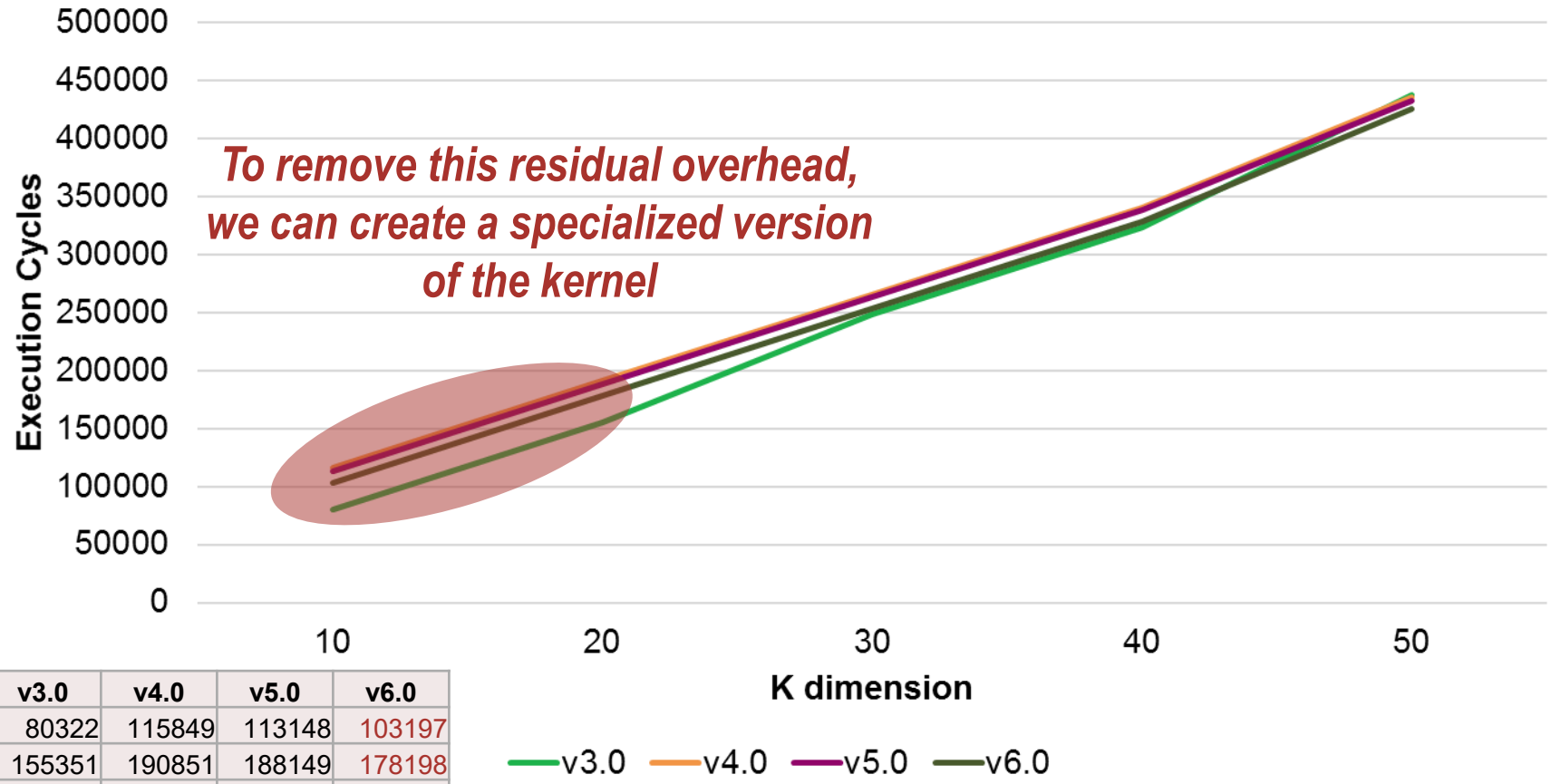
Matrix multiplication v6.0

```
void matmul(int *__restrict__ A, int *__restrict__ B, int *__restrict__ C,
            int M1, int N1, int K1) {
  int idx_A1 = 0, idx_A2 = 1;
  int i = 0;
  do
  {
    int j = 0;
    do
    {
      int idx_B1 = j, idx_B2 = N1+j;
      int val = 0;
      int k = 0;
      do
      {
         int A1 = A[idx_A1+2*k], A2 = A[idx_A2+2*k];
         int B1 = B[idx_B1], B2 = B[idx_B2];
         idx_B1 += N1; idx_B2 += N1;
         asm volatile(""::::"memory");
         val += A1 * B1;
         val += A2 * B2;
         k++;
      } while(k < K1/2);
      C[i*N1+j] = val;
      j++;
    } while(j < N1);
    idx_A1 += K1; idx_A2 += K1;
    i++;
  } while (i < M1);
}
```

Assembly v6.0

```
...
1c0084b4:       01c5407b                lp.setup        x0,a0,1c0084ec
1c0084b8:       4f05                    li      t5,1
1c0084ba:       00c98eb3                add     t4,s3,a2
1c0084be:       8e32                    mv      t3,a2
1c0084c0:       834a                    mv      t1,s2
1c0084c2:       88a6                    mv      a7,s1
1c0084c4:       4781                    li      a5,0
1c0084c6:       05ea6833                p.max   a6,s4,t5
1c0084ca:       0001                    nop
1c0084cc:       00c840fb                lp.setup        x1,a6,1c0084e4
1c0084d0:       0088a38b                p.lw    t2,8(a7!)
1c0084d4:       00832f8b                p.lw    t6,8(t1!)
1c0084d8:       213e728b                p.lw    t0,s3(t3!)
1c0084dc:       213eff0b                p.lw    t5,s3(t4!)
1c0084e0:       425387b3                p.mac   a5,t2,t0
1c0084e4:       43ef87b3                p.mac   a5,t6,t5
1c0084e8:       00f4222b                p.sw    a5,4(s0!)
...
```


Experimental results v6.0



K	v1.0	v2.0	v3.0	v4.0	v5.0	v6.0
10	80778	105318	80322	115849	113148	103197
20	227970	205347	155351	190851	188149	178198
30	327970	355535	248035	265851	263149	253198
40	427970	468035	323035	340851	338149	328198
50	555812	623109	437120	435922	432971	425264

Issue #2: Change the type of iteration variables

- Suppose to use uint16_t iteration variables (i, j, k) in v6.0

V6.0, M=50, N=50, K=50

[0] cycles = 425264
[0] instr = 403094
[0] active cycles = 425264
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 0
[0] imiss = 2953

V6.0, M=50, N=50, K=50 (uint16_t vars)

[0] cycles = 1100850
[0] instr = 907888
[0] active cycles = 1100850
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 0
[0] imiss = 60049

- SOLUTION: Avoid it!!!

Assembly (issue #2)

```
...
1c008aee:       9f3a            add     t5,t5,a4
1c008af0:       43a30e33        p.mac   t3,t1,s10
1c008af4:       0885            addi    a7,a7,1
1c008af6:       1008d8b3        p.exthz a7,a7
1c008afa:       43980e33        p.mac   t3,a6,s9
1c008afe:       fc98c9e3        blt     a7,s1,1c008ad0 <matmul+0x56>
1c008b02:       04098863        beqz    s3,1c008b52 <matmul+0xd8>
1c008b06:       0f0a            slli    t5,t5,0x2
1c008b08:       21e5ff03        p.lw    t5,t5(a1)
1c008b0c:       000aa883        lw      a7,0(s5)
1c008b10:       00590833        add     a6,s2,t0
1c008b14:       080a            slli    a6,a6,0x2
1c008b16:       9f46            add     t5,t5,a7
1c008b18:       9e7a            add     t3,t3,t5
1c008b1a:       0285            addi    t0,t0,1
1c008b1c:       01c66823        p.sw    t3,a6(a2)
1c008b20:       1002d2b3        p.exthz t0,t0
1c008b24:       f8e2cce3        blt     t0,a4,1c008abc <matmul+0x42>
1c008b28:       0a05            addi    s4,s4,1
1c008b2a:       100a5a33        p.exthz s4,s4
1c008b2e:       943e            add     s0,s0,a5
1c008b30:       9ade            add     s5,s5,s7
1c008b32:       83da            mv      t2,s6
1c008b34:       f6da4fe3        blt     s4,a3,1c008ab2 <matmul+0x38>
...
```


Issue #3: Boundary check on internal loops

- Adding a boundary check to the inner loop:

```
if(K1 > 0)
  do  // inner loop (3rd level)
  {
  …
  } while(k<K1/2);
```

V6.0, M=50, N=50, K=50
[0] cycles = 425264
[0] instr = 403094
[0] active cycles = 425264
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 0
[0] imiss = 2953

V6.0, M=50, N=50, K=50 (boundary check)
[0] cycles = 432949
[0] instr = 413045
[0] active cycles = 432949
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 0
[0] imiss = 695

Boundary check on internal loops

- We can move the condition to the outer loops
  - We need to add code for the else case
  - We get the same performance of the previous case

```
if (K1>0)
  do // outer loop (1st level)
  {
   …
  } while (i < M1);
else
  do
  {
    C[i] = 0;
    i++;
  } while (i < M1*N1);
```

V6.0, M=50, N=50, K=50 (boundary check)

[0] cycles = 425264
[0] instr = 403094
[0] active cycles = 425264
[0] ext load = 0
[0] TCDM cont = 0
[0] ld stall = 0
[0] imiss = 2953

Link time optimization (LTO)

- Link Time Optimization (LTO) outputs the GCC internal representation (GIMPLE) into an ELF section of the object file with the aim to optimize compilation units as a single module
- To enable LTO add -flto to both compiler and linker flags
- Options:
  - -flto-partition=*lto1*|*max*|*balanced*
  - -flto-compression-level=0..9
- When is ok to use LTO? Basically, always!

A checklist to avoid common mistakes

- Remove all warnings (avoid -Wall)
- Double-check the march parameter in the compilation line
- Verify that floating-point arithmetic is used properly
  - Avoid call to emulation routines
  - Use the -mtune flag for better instruction scheduling (experimental)
- Verify data allocation when moving execution from fabric controller to cluster cores
- Apply parallelization techniques to optimized code
  - When the code is really optimized, the speedup could be drastically reduced