Mastering the PULP GCC toolchain

Introduction to the compilation toolchain and Performance-driven optimization techniques

Giuseppe Tagliavini <giuseppe.tagliavini@unibo.it>
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](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:
     
     ```bash
     git clone --recursive https://github.com/pulp-platform/pulp-riscv-gnu-toolchain
     ```
  3. Configure:
     ```bash
     ./configure --prefix=<INSTALL_DIR> --with-arch=rv32imc --with-cmodel=medlow --enable-multilib
     ```
  4. Add the bin directory to the path variable:
     ```bash
     export PATH=<INSTALL_DIR>/bin:$PATH
     ```
  5. Build the toolchain:
     ```bash
     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

- `-mcmmodel=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

- `-mcmmodel=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 number of instructions 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
Mastering the PULP GCC toolchain

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
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  \rightarrow multiply-and-accumulate (MAC)
Space complexity  \rightarrow O(M \cdot N)
Time complexity  \rightarrow 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 \( \Rightarrow \) 2 lw + 1 MAC + 1 sw \( \Rightarrow \) \( M*N*K *4 \)
  - 1 elementary operation \( \Rightarrow \) 2 lw + 1 MAC + (1/K) sw \( \Rightarrow \) \( M*N*K *3 + N*M \)

- 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

```c
#define M 50
#define N 50
#define K 30
```

```assembly
...  
0168407b lp.setup x0,a6,1c008716
41ce87b3 sub a5,t4,t3
17f1 addi a5,a5,-4
8389 srli a5,a5,0x2
0008a22b p.sw zero,4(0x7)
861a mv a2,t1
86f2 mv a3,t3
4701 li a4,0
0785 addi a5,a5,1
0067c0fb lp.setup x1,a5,1c00870e
0046a50b p.lw a0,4(a3!)
0c86258b p.lw a1,200(a2!)
42b50733 p.mac a4,a0,a1
fee8ae23 sw a4,-4(0x7)
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

1. Declare variables

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

2. Repeat measures in a loop

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

3. Start measuring

```c
BEGIN_STATS_LOOP();
START_STATS();
matmul(A, B, C, M, N, K);
STOP_STATS();
```

4. Pause measuring (accumulate into variables)

5. End of measuring loop

```c
END_STATS_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] Id 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] Id stall = 50000
[0] imiss = 0
Assembly (v1.0 M=50 N=50 K=10)

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 -mhwlooppalign flag

<table>
<thead>
<tr>
<th>V1.0, M=50, N=50, K=10</th>
<th>V1.0, M=50, N=50, K=10 -mhwlooppalign</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] cycles = 80778</td>
<td>[0] cycles = 78330</td>
</tr>
<tr>
<td>[0] instr = 53326</td>
<td>[0] instr = 53327</td>
</tr>
<tr>
<td>[0] active cycles = 80778</td>
<td>[0] active cycles = 78330</td>
</tr>
<tr>
<td>[0] ext load = 0</td>
<td>[0] ext load = 0</td>
</tr>
<tr>
<td>[0] TCDM cont = 0</td>
<td>[0] TCDM cont = 0</td>
</tr>
<tr>
<td>[0] ld stall = 25000</td>
<td>[0] ld stall = 25000</td>
</tr>
<tr>
<td>[0] imiss = 2449</td>
<td>[0] imiss = 0</td>
</tr>
</tbody>
</table>
Anomaly #2

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

<table>
<thead>
<tr>
<th>V1.0, M=50, N=50, K=40</th>
<th>V1.0, M=50, N=50, K=50</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] cycles = 427970</td>
<td>[0] cycles = 554784</td>
</tr>
<tr>
<td>[0] instr = 327869</td>
<td>[0] instr = 403024</td>
</tr>
<tr>
<td>[0] active cycles = 427970</td>
<td>[0] active cycles = 554784</td>
</tr>
<tr>
<td>[0] ext load = 0</td>
<td>[0] ext load = 0</td>
</tr>
<tr>
<td>[0] TCDM cont = 0</td>
<td>[0] TCDM cont = 0</td>
</tr>
<tr>
<td>[0] Id stall = 100000</td>
<td>[0] Id stall = 125000</td>
</tr>
<tr>
<td>[0] imiss = 0</td>
<td>[0] imiss = 127114</td>
</tr>
</tbody>
</table>
Removing anomaly #2

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

<table>
<thead>
<tr>
<th>V1.0, M=50, N=50, K=50</th>
<th>V1.0, M=50, N=50, K=50 -mhwloopalign</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] cycles = 554784</td>
<td>[0] cycles = 543175</td>
</tr>
<tr>
<td>[0] instr = 403024</td>
<td>[0] instr = 403074</td>
</tr>
<tr>
<td>[0] active cycles = 554784</td>
<td>[0] active cycles = 543175</td>
</tr>
<tr>
<td>[0] ext load = 0</td>
<td>[0] ext load = 0</td>
</tr>
<tr>
<td>[0] TCDM cont = 0</td>
<td>[0] TCDM cont = 0</td>
</tr>
<tr>
<td>[0] ld stall = 125000</td>
<td>[0] ld stall = 125000</td>
</tr>
<tr>
<td>[0] <strong>imiss = 127114</strong></td>
<td>[0] <strong>imiss = 7300</strong></td>
</tr>
</tbody>
</table>
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
}
## Measuring performance of v2.0

- We have not removed all the stalls. Why?

<table>
<thead>
<tr>
<th>V2.0, M=50, N=50, K=10</th>
<th>V2.0, M=50, N=50, K=20</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] cycles = 78329</td>
<td>[0] cycles = 266789</td>
</tr>
<tr>
<td>[0] instr = 53326</td>
<td>[0] instr = 194139</td>
</tr>
<tr>
<td>[0] active cycles = 78329</td>
<td>[0] active cycles = 266789</td>
</tr>
<tr>
<td>[0] ext load = 0</td>
<td>[0] ext load = 0</td>
</tr>
<tr>
<td>[0] TCDM cont = 0</td>
<td>[0] TCDM cont = 0</td>
</tr>
<tr>
<td>[0] ld stall = 25000</td>
<td>[0] ld stall = 72500</td>
</tr>
<tr>
<td>[0] imiss = 0</td>
<td>[0] imiss = 49</td>
</tr>
</tbody>
</table>
Assembly v2.0 (M=50 N=50 K=50)

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

```assembly
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

Load stall

Writing back to memory in the inner loop
```
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: `-fpassname`
  - To remove an optimization pass: `-fno-passname`

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

```c
/* 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_evolution (loop, niter->assumptions);
    niter->may_be_zero = simplify_using_outer_evolution (loop, niter->may_be_zero);
    niter->niter = simplify_using_outer_evolution (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:

```c
#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:

  ```c
  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:

  ```c
  volatile int m = M;
  volatile int n = N;
  volatile int k = K;
  matmul(A, B, C, m, n, k);
  ```
Matrix multiplication v4.0

```c
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

<table>
<thead>
<tr>
<th>K</th>
<th>v1.0</th>
<th>v2.0</th>
<th>v3.0</th>
<th>v4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>80778</td>
<td>105318</td>
<td>80322</td>
<td>115849</td>
</tr>
<tr>
<td>20</td>
<td>227970</td>
<td>205347</td>
<td>155351</td>
<td>190851</td>
</tr>
<tr>
<td>30</td>
<td>327970</td>
<td>355535</td>
<td>248035</td>
<td>265851</td>
</tr>
<tr>
<td>40</td>
<td>427970</td>
<td>468035</td>
<td>323035</td>
<td>340851</td>
</tr>
<tr>
<td>50</td>
<td>555812</td>
<td>623109</td>
<td>437120</td>
<td>435922</td>
</tr>
</tbody>
</table>
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       srl   a4,a4,0x1 
1c008110: 00160313    addi  t1,a2,1 
1c008114: 8f0e09      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!) 
...
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
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);
}
Mastering the PULP GCC toolchain

Assembly v5.0

... lp.setup x0,t2,1c008130
1c0080ec: 0223c07b
1c0080f0: fff60713
1c0080f4: 8305
1c0080f6: 00160e13
1c0080fa: 4e89
1c0080fc: 00598833
1c008100: 86a6
1c008102: 8896
1c008104: 834a
1c008106: 4781
1c008108: 0705
1c00810a: 05de4563
1c00810e: 0001
1c008110: 00c740fb
1c008114: 00832f8b
1c008118: 0086ae8b
1c00811c: 2188ff0b
1c008120: 2188e0b
1c008124: 43ef87b3
1c008128: 43ce87b3
1c00812c: 00f4222b
...  
-1 branch + 1 nop!!!

lp.setup x1,a4,1c008128
1c008114: 00832f8b
1c008118: 0086ae8b
1c00811c: 2188ff0b
1c008120: 2188e0b
1c008124: 43ef87b3
1c008128: 43ce87b3
1c00812c: 00f4222b
...
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)
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

```
1c0084bc: 01c5407b
1c0084b8: 4f05
1c0084ba: 00c98eb3
1c0084be: 8e32
1c0084c0: 834a
1c0084c2: 88a6
1c0084c4: 4781
1c0084c6: 05ea6833
1c0084ca: 0001
1c0084cc: 00c840fb
1c0084d0: 0088a38b
1c0084d4: 00832f8b
1c0084d8: 213e728b
1c0084dc: 213eff0b
1c0084e0: 425387b3
1c0084e4: 43ef87b3
1c0084e8: 00f4222b

lp.setup x0,a0,1c0084ec
li t5,1
add t4,s3,a2
mv t3,a2
mv t1,s2
mv a7,s1
li a5,0
p.max a6,s4,t5
nop
lp.setup x1,a6,1c0084e4
p.lw t2,8(a7!)
p.lw t6,8(t1!)
p.lw t0,s3(t3!)
p.lw t5,s3(t4!)
p.mac a5,t2,t0
p.mac a5,t6,t5
p.sw a5,4(s0!)
```
To remove this residual overhead, we can create a specialized version of the kernel.
Issue #2: Change the type of iteration variables

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

**SOLUTION: Avoid it!!!**

<table>
<thead>
<tr>
<th>V6.0, M=50, N=50, K=50</th>
<th>V6.0, M=50, N=50, K=50 (uint16_t vars)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] cycles = 425264</td>
<td>[0] cycles = 1100850</td>
</tr>
<tr>
<td>[0] instr = 403094</td>
<td>[0] instr = 907888</td>
</tr>
<tr>
<td>[0] active cycles = 425264</td>
<td>[0] active cycles = 1100850</td>
</tr>
<tr>
<td>[0] ext load = 0</td>
<td>[0] ext load = 0</td>
</tr>
<tr>
<td>[0] TCDM cont = 0</td>
<td>[0] TCDM cont = 0</td>
</tr>
<tr>
<td>[0] ld stall = 0</td>
<td>[0] ld stall = 0</td>
</tr>
<tr>
<td>[0] imiss = 2953</td>
<td>[0] imiss = 60049</td>
</tr>
</tbody>
</table>
Assembly (issue #2)

...
Issue #3: Boundary check on internal loops

- Adding a boundary check to the inner loop:

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

<table>
<thead>
<tr>
<th>V6.0, M=50, N=50, K=50</th>
<th>V6.0, M=50, N=50, K=50 (boundary check)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] cycles = 425264</td>
<td>[0] cycles = 432949</td>
</tr>
<tr>
<td>[0] instr = 403094</td>
<td>[0] instr = 413045</td>
</tr>
<tr>
<td>[0] active cycles = 425264</td>
<td>[0] active cycles = 432949</td>
</tr>
<tr>
<td>[0] ext load = 0</td>
<td>[0] ext load = 0</td>
</tr>
<tr>
<td>[0] TCDM cont = 0</td>
<td>[0] TCDM cont = 0</td>
</tr>
<tr>
<td>[0] ld stall = 0</td>
<td>[0] ld stall = 0</td>
</tr>
<tr>
<td>[0] imiss = 2953</td>
<td>[0] imiss = 695</td>
</tr>
</tbody>
</table>
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