EE6470 ESL Design and Synthesis

Final report

0/1 Knapsack by parallel branch and bound

by patata0717

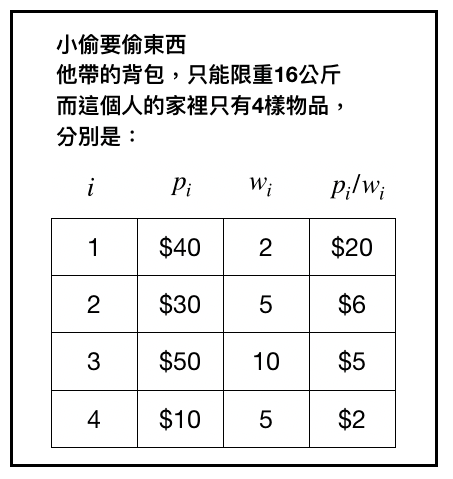
https://github.com/patata0717/EE6470-ESL-design-and-synthesis/tree/main/final

1. Introduction

In this final project, we will choose an algorithm, first use HLS and try for different HLS config, and port it to riscv-vp and try to implement multi-core, and compare with single-core.

1. 0/1 Knapsack problem by parallel branch and bound

0/1 knapsack problem is a classic optimization problem. Branch and bound is an useful technique to reduce try effort, and it can be parallelled. This technique is very useful for Integer Linear Programming solver.



gold

silver

copper

comp

A thief has a bag of weight limit 16, and has 4 items to choose from, each item has different value and weight, it can only be taken(1) or not taken(0). The objective is to find a combination of items that within weight limit and gives the max value sum.

One of the method to solve this is to build a tree of 2^4 leaf nodes, and try all possible combinations. But we can utilize branch and bound to stop the branch early.

In our 100 items testcase, for brute force needs to try 2^100=1.2×1030，B&B only needs to try 223 nodes.

We need to calculate the Upper bound and the Lower bound for each node, and keep a “best value” globally. LB will update the “best value”. If each node’s UB is smaller than current best value, it will be pruned because it has no chance to gives the better result.

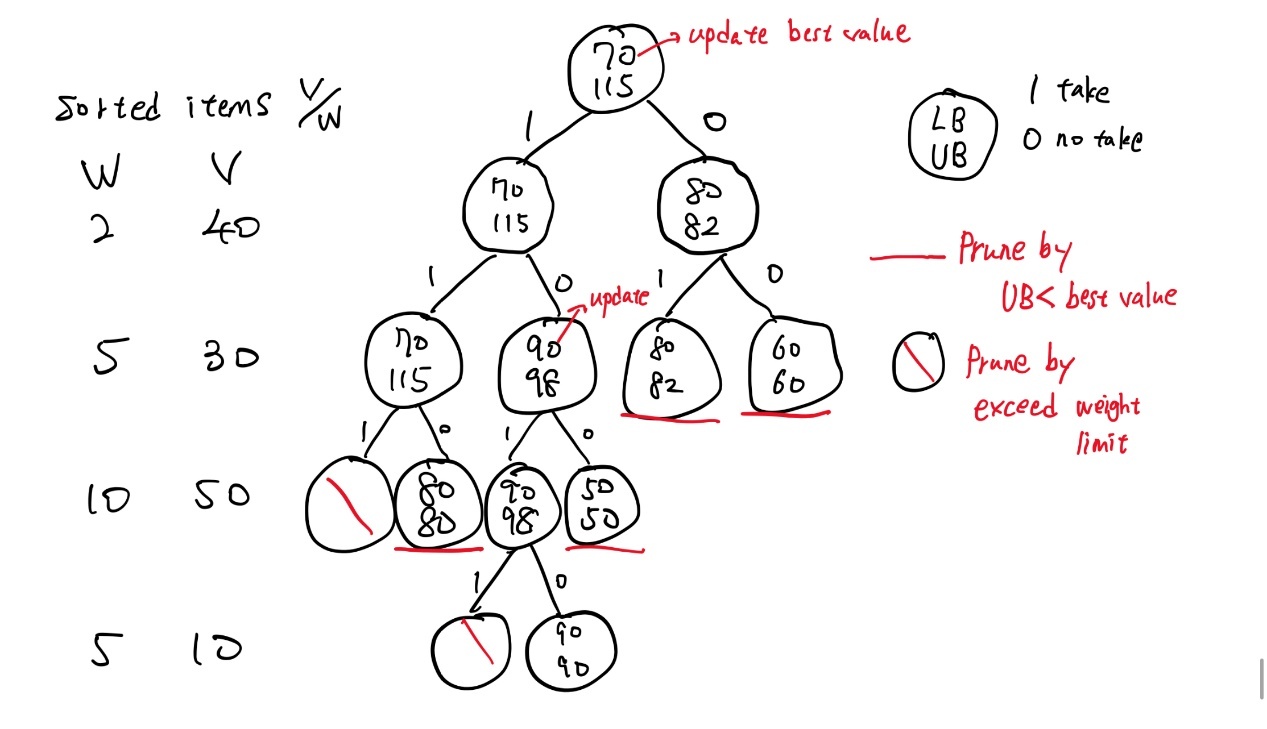
Because each branch is independent, we can exploit parallelism.

For exploring nodes, we can use DFS, BFS, or other strategies. Parallel B&B is a BFS way for searching.

We first sort all item by V/W, so we will take the more valuable item first.

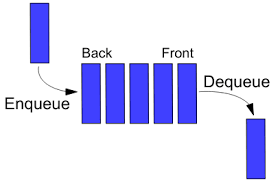
The LB is defined as “greedily filling the knapsack with the remaining items until the capacity is about to reached or exceeded”

The UB is defined as “LB + ratio cut”. Which means, if current item is “copper” of weight 10, value 50, but my remaining bag size is only 9, The thief can use a knife and cut 9/10 of it, which get the value 45.



Node can be pruned by “UB < best value” or “exceed weight limit”. In the above example, at last, we can find LB=90, so best value is 90.

For parallel B&B, we construct a “global queue”, when we go to each node, we face 2 possibilities(next item take or no take), so we push 2 nodes into queue. And we keep dispatch node to PE once any PE idle, and PE will calculate UP and LB for CPU, which is the main computation for the algorithm, thus we can exploit parallelism. Processing nodes multiple times faster than single PE.



Each node push 2 possibilities queue

Dispatch node to PEs

1. Part2: riscv-vp multicore
   1. Outline

My design is a 1 CPU multi-PE design, CPU act as SW, PE act as accelerator. CPU can keep dispatching idle PEs, and each PE can process parallelly.

We need to let PE access the whole item table for calculating LB and UP, it is too costly to transfer the whole table every time, so we first fill the item table in PE, and then start performing node dispatch.

So for multi-PEs, we need to copy item table to each PE.

First stage: CPU transfer header(num of items), PE define its memory

Second stage: CPU fill the item table of PE

Third stage: CPU keep dispatching node, and get result from PEs

* 1. Develop

I choose tiny32, which is a very simple single core vp (the basic-acc is too complex), which has SimpleBus and GDB debug support, I try to add DMA, PLIC on it for supporting interrupt request(IRQ).

I move “fe310\_plic”, “dma.h” from basic-acc to tiny32.

Add DMA, PLIC, PEs to SimpleBus.

SimpleBus<3, 9>

Initiator:

1. Core\_mem\_if
2. Gdb\_if
3. DMA

Target:

1. Mem
2. Clint
3. Sys
4. Dma
5. Plic
6. PE0
7. PE1
8. PE2
9. PE3
   1. IRQ handling

I copy from pi-multicore method, in SW, an init() function to register\_interrupt\_handler, and put init() in the first line of main(), when I call main() it will call init(). So that we can identify all IRQ calls and write a handler to handle them.

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

AI 產生的內容可能不正確。

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

AI 產生的內容可能不正確。

We can see that, we first register all IRQ, and for write\_data\_to\_ACC(), it follows by an irq=4(DMA) means DMA write completed.

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

AI 產生的內容可能不正確。

In this picture, we can see that we have IRQ for DMA(4) and PE3(8), PE0(5), and they will call irq\_PE() or dma\_irq\_handler() for CPU to handle the situation.

As we can see, CPU dispatch and PE processing runs concurrently.

DMA will block other read or write by “dma\_busy”.

PE will block other dispatch by “pe\_idle”.

For a dispatch, when it finished write\_data\_to\_ACC(), it complete, and start to dispatching new node.

* 1. Memory map

一張含有 文字, 螢幕擷取畫面, 設計 的圖片

AI 產生的內容可能不正確。

Original tiny32+ DMA+PLIC+PEs

* 1. DMA transfer—4 byte node datatype

一張含有 文字, 螢幕擷取畫面, 字型, 功能表 的圖片

AI 產生的內容可能不正確。

This is the node datatype, which is 32 bits raw data, which can be read in

(value, weight, depth) or (UB, LB, id). The DMA len is 4 bytes, so I allocate 12 bits for each V and W, max 4096, and 8 bits for depth, max 256. So it can’t handle too large testcase. My largest testcase is 35 items.

We send node in (V, W, D) and get result in (UB, LB, id).

* 1. Identify the return node.

The return type is (UB, LB), we need to know its original V, W, D information for dispatching next node. But because it is multicore, the node return order is not guaranteed, we need to have a method to identify the return node. Luckily, it is guaranteed that for same PE, the return node is guaranteed in order. So we can record id for same PE, and build a table for backup before sending, then we can look up the table to get original V, W, D for returning node.

**Identify the returned node**

id[pe]←identify by irq [order]←given value by PE

PE1’s 6th process node—id[2][5]

PE1’s 7th process node—id[2][6]

It is guaranteed that id[2][5] will arrive earlier than id[2][6].



With these 2 index, we can look up the backup table for getting (V, W, D)

* 1. PE process speed should match DMA and PLIC

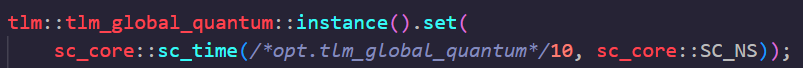
The biggest issue I faced, is I found that PE processing looks like “block” the interrupt. When calling interrupt, my CPU irq handler will wait until PE processing ends, and then dispatching next PE, which is not match my parallel design(imagine when PE is processing, I can’t dispatch new PE until the process ends). I came up 2 solutions:

1. Raise priority of DMA(I thought maybe DMA irq is blocked by PE’s) in bootstrap.S

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

AI 產生的內容可能不正確。

1. Lower global quantum(vp/tiny32/main.cpp)



And I found that the smallest quantum can be set as 10ns, which is the ISS cycle, if lower than this, it will pop error.

Method 1 has no effect, so it is not PLIC overlap issue. But by lower the quantum and keep experimenting, I found that I should set the PE processing time at the scale similar to DMA and PLIC, which is around 100000ns. If PE process too fast, it ends way faster than DMA and PLIC, which looks like it “block” them.

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

AI 產生的內容可能不正確。

By adding delay in PE, it finally looks like PE process is running concurrently.

* 1. Do not call printf() while PE is in wait()

CPU shouldn’t call printf() while PE’s is processing, for CPU can break the wait() in PE and cause error. If you really want to see the debug print, please lower the wait() below 10000ns, and it is save to call printf() in CPU.

* 1. Can’t raise PE delay higher than 200000ns

It will pop this

一張含有 文字, 字型, 螢幕擷取畫面, 白色 的圖片

AI 產生的內容可能不正確。

The addr 2095256(0x001FF898) is inside CPU mem(0x00000000~0x02000000), I also try to use larger CPU mem, but still can’t route addr 2095256, so I have no idea why cause this issue.

* 1. Dispatch loop and results

一張含有 文字, 螢幕擷取畫面, 陳列, 字型 的圖片

AI 產生的內容可能不正確。

The while\_1 loop will keep finding idle PE and dispatch them. If all PE are busy, it “wfi”; If all idle && q empty(), loop ends. Otherwise, we will loop it again and find idle PE.



一張含有 文字, 字型, 螢幕擷取畫面, 圖形 的圖片

AI 產生的內容可能不正確。

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

AI 產生的內容可能不正確。

* 1. DMA calculation

Each node 4 byte

Fill item table: Size × num of PEs × (write data, write cmd), consecutive

For item35 example, size=35\*4 bytes, 4 PEs

(4+(35\*2-1)\*4/4)\*4 = 292 cycles

For item4 example, size=4\*4 bytes, 4 PEs

(4+(4\*2-1)\*4/4)\*4 = 44 cycles

Run: num of dispatch × (write data, write cmd, read data), non consecutive

In above size35 example, num of dispatch = 183, 4 PE(47+46+45+45)

(4+1+4)\* (47+46+45+45) = 1647 cycles

In above size4 example, num of dispatch = 13, 4 PE(4+3+3+3)

(4+1+4)\* (4+3+3+3) = 117 cycles

We can estimate that, DMA takes 70%~85% of time in dispatch. 15%~30% for fill item table.

* 1. Experiment

Setting: ISS speed: 10ns, global quantum=10ns

Stimulus size: 35, PE delay: 150000 ns

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1core | 2core | 3core | 4core |
| Cycle(ns/10) | 7391854 | 6695029 | 8391463 | 8907279 |
| Num\_instr | 1820401 | 2092089 | 2470684 | 2797200 |
| PE1 dispatch | 183 | 92 | 62 | 47 |
| PE2 dispatch |  | 91 | 61 | 46 |
| PE3 dispatch |  |  | 60 | 45 |
| PE4 dispatch |  |  |  | 45 |

Stimulus size: 35, PE delay: **1500** ns

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1core | 2core | 3core | 4core |
| Cycle(ns/10) | 5583620 | 6685265 | 7790507 | 8897203 |
| Num\_instr | 1740188 | 2091656 | 2443903 | 2796609 |
| PE1 dispatch | 183 | 92 | 61 | 46 |
| PE2 dispatch |  | 91 | 61 | 46 |
| PE3 dispatch |  |  | 61 | 46 |
| PE4 dispatch |  |  |  | 45 |

Stimulus size: 20, PE delay: 150000 ns

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1core | 2core | 3core | 4core |
| Cycle(ns/10) | 2116169 | 2416120 | 3191621 | 3728466 |
| Num\_instr | 564422 | 757775 | 972592 | 1176222 |
| PE1 dispatch | 37 | 19 | 13 | 10 |
| PE2 dispatch |  | 18 | 12 | 9 |
| PE3 dispatch |  |  | 12 | 9 |
| PE4 dispatch |  |  |  | 9 |

Stimulus size: 4, PE delay: 150000 ns

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1core | 2core | 3core | 4core |
| Cycle(ns/10) | 633190 | 723824 | 943960 | 1083968 |
| Num\_instr | 173899 | 225873 | 285298 | 340813 |
| PE1 dispatch | 13 | 7 | 5 | 4 |
| PE2 dispatch |  | 6 | 4 | 3 |
| PE3 dispatch |  |  | 4 | 3 |
| PE4 dispatch |  |  |  | 3 |

Several facts found:

1. **Cycles will increase while # of PEs increase**

This is very weird for me, in my above calculation, the number of DMA transfer is the same for # of PEs, but PE calculation can be parallel.

My guess is maybe small # of PEs can utilize DMA well, and take less cycle for DMA. Or there might be some calculations is not negligible for large # of PEs, such as PLIC operation.

1. **Reduce PE delay can improve small # of PE but can’t improve large # of PE**

Because PE calculation is in parallel, so single PE benefit the most, but it doesn’t explain why it didn’t improve 4 PE. Delay reduce from 150000ns→1500ns，and each PE processing 40~50 nodes, but the improve only 100000 ns, which I can’t explain.

1. **Cycle grows with num of dispatch, and grows slower when test case increase.**

It’s logical, there are some operation is not propotional to # of dispatch.

1. Part1: HLS
   1. Constrain latency using wait()

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

AI 產生的內容可能不正確。

Use simple wait() for each put() and get(), which stands for 1ns.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| items | 4 | 20 | 35 | 100 |
| node visit | 13 | 37 | 183 | 233 |
| Cycle(ns)(BEH) | 320 | 960 | 4030 | 5680 |

* 1. Loops

In div\_u16\_u8: **div\_loop**, for shift and subtract (16 bits)

In bound\_kernel: **pack\_loop**, when calculating LB, take all rest item until bag full (items size)

In thread1: **read\_loop**, fill item table, (items size)

* 1. Pipeline

**while\_1**, which is the main dispatch-handle while1 loop, contain function calls, which has another layer of loop in function.

I flatten the function call to main, so that it won’t pop error.

* 1. Coding style improved
* Trim port bit dut input<56>, output<32>, to input<40>, output<32>.

I originally put header in input, so its (8+8)header+(V16+W16+D8), and I make header shared the bit with items.

But have no effect in improving synthesis area. I guess is because compiler will automatically trim it to max bandwidth needed, which is 40.

* Div by shift and subtract

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

AI 產生的內容可能不正確。

Which is a good way for div, slower than LUT but save memory.

* 1. Experiment

Item size=100 (main memory will be 100\*16\*2 in DUT)

(will be map to reg bank or RAM 8×8 or arrary(buf))

+flatten all array +dpopt=op

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Seq | Comb | BB | Total | Cycle(ns) |
| REG | 9581.5(1751) | 15438.3 | 0.0 | 25019.8 | 65095 |
| REG\_UNROLL | 9647.1(1763) | 17230.4 | 0.0 | 26877.5 | 52615 |
| REG\_UNROLL\_  FAST | 9526.8(1741) | 18879.0 | 0.0 | 28405.8 | 50495 |
| MEM | 837.2(153) | 2872.9 | 2000.0 | 5710.1 | 108865 |
| MEM\_UNROLL | 870.0(159) | 4229.1 | 2000.0 | 7099.2 | 97340 |
| MEM\_UNROLL\_  FAST | 875.5(160) | 5149.3 | 2000.0 | 8024.9 | 138010 |
| BUF | 10117.7(1849) | 18990.2 | 0.0 | 29107.9 | 65095 |
| BUF\_UNROLL | 10188.9(1862) | 22233.6 | 0.0 | 32422.5 | 50495 |
| BUF\_UNROLL\_  FAST | 10090.4(1844) | 22394.9 | 0.0 | 32485.3 | 50495 |

* Cycle will reduce for UNROLL
* Cycle will not necessarily reduce for UNROLL\_FAST, for MEM, it significantly increases.
* Area will increase for UNROLL and FAST

Item size=20:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Seq | Comb | BB | Total | Cycle(ns) |
| REG | 2577.3(471) | 5574.0 | 0.0 | 8151.3 | 5160 |
| REG\_UNROLL | 2648.4(484) | 6553.6 | 0.0 | 9202.0 | 3000 |
| REG\_UNROLL\_  FAST | 2643.0(483) | 8274.2 | 0.0 | 10917.2 | 2640 |
| MEM | 837.2(153) | 2860.5 | 2000.0 | 5697.7 | 6665 |

* For smaller input size, UNROLL and UNROLL\_FAST still can improve speed but increase area.
* Block memory didn’t reduce for smaller input size

PIPELINE includes unroll loops, and map mem to reg bank.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Seq | Comb | BB | Total | Cycle(ns) |
| PIPELINE, II=1 | fail | | | | |
| PIPELINE, II=2 | 20946.8(3828) | 52766.5 | 0.0 | 73713.3 | 25485 |
| PIPELINE, II=3 | 19283.3(3524) | 50286.8 | 0.0 | 69570.1 | 28980 |
| PIPELINE, II=4 | 18829.2(3441) | 46444.2 | 0.0 | 65273.3 | 31310 |

* Pipeline significantly reduced cycles(~50%), but increase area(+~100%)
* With smaller II, reduced more cycles and increase more area.
  1. Add HLS and riscv-vp together

RISCV-vp is 8-digit, HLS is 5-digit, RISCV-vp will dominate HLS lantency.