ADVANCED COMPUTER ARCHITECTURE - HOMEWORK III

Student: Francisco Javier Blázquez Martínez
E-mail: francisco.blazquezmartinez@epfl.ch
Sciper: 331229

Vivaldo HLS synthesis, optimization and performance and resources analysis of different kernels.
Report to be used as compliment to the Vivaldo HLS projects.

Kernel 1.-

1. **Kernel 1 synthesis:** *See /kernel1/prj_kernel1/solution1/syn/report/kernel1_csynth.rpt*
2. **Optimization directives used:** Pipeline
   See /kernel1/kernel1.cpp
   See /kernel1/prj_kernel1/solution2/syn/report/kernel1_csynth.rpt
3. **Best initiation interval:** 1
4. **Optimizations:** Loop pipelining
   Result: Latency halved, initiation interval reached
5. **Resources requeriments comparison:** Both solutions require essentially the same hardware. Solution 2 uses less registers but in this we are reading from memory at every cycle when in solution 1 memory is available for reading the half of the time.
   With loop unrolling we obtain an smaller clock period but the same latency, this is because we have some bottlenecks among our hardware resources, in our case, the memory. Without these limitations, we could create hardware with a smaller latency since the loop iterations are independent and can be executed with the desired level of parallelism.

Kernel 2.-

1. **Kernel 2 synthesis:** *See /kernel2/prj_kernel2/solution1/syn/report/kernel2_csynth.rpt*
2. **Optimization directives used:** Pipeline
   See /kernel2/kernel2.cpp
   See /kernel2/prj_kernel2/solution5/syn/report/kernel2_csynth.rpt
3. **Best initiation interval:** 1
4. **Optimizations:** Loop pipelining and explicit management of loop-carry dependencies.
   Direct loop pipelining has some latency reduction but it's improvable.
   Loop unrolling detects memory dependencies between iterations but exceeds the resouces.
   Explicit declaration of *Dependence* directive for the array variable also fails to detect data reuse so auxiliar variables and explicit loop-carry dependencies management was required.
   Result: Latency reduced, initiation interval 1 reached
5. **Resources requeriments comparison:** The main difference is the memory usage. For our final solution, thanks to data reusage we do not read anything from memory inside of our loop. This makes our solution much faster. Also, as our final solution is pipelined, the usage of hardware for computing the expression inside the loop (ADD, MUL) is more intense, taking place at every single cycle.
   At this point I thought on a parallel hardware execution in some way as trying to compute two elements of the array per cycle. Again the memory here was a bottleneck because we can only write at one address each cycle (what is already happening with our current best solution) and, even more, looking at the schedule and our critical path time we can not add more operations respecting the fixed clock rate.

Kernel 3.-

1. **Kernel 3 synthesis:** See /kernel3/prj_kernel3/solution1/syn/report/kernel3_csynth.rpt
2. **Optimization directives used:** Pipeline, Array partition
   See /kernel3/kernel3.cpp
   See /kernel3/prj_kernel3/solution4/syn/report/kernel3_csynth.rpt
3. **Best initiation interval:** 5
   Initiation interval one is not achievable because the array *hist* is not accessed sequentially but in an arbitrary order given by *index*. This implies that we might need the result of the previous iteration for the current one and this result, as we are operating with float, requires four cycles.
4. **Optimizations:** Loop pipelining and Array partition
   Based on the previous observation, loop pipelining or unrolling is not enough to obtain a good initiation interval. Array partition reduces loop's latency and initation interval thanks to avoiding memory reads.
   Result: Latency reduced about 35%
5. **Resources requeriments comparison:** The main difference is the memory and registers usage because, after doing complete partition of the array *hist*, it's stored in registers that can be accessed much faster and simultaneously. This makes that every iteration we only have to read *weight[i]* and *inex[i]* but of course the number of register used is around 1024.

Kernel 4.-

1. **Kernel 4 synthesis:** See /kernel4/prj_kernel4/solution1/syn/report/kernel4_csynth.rpt
2. **Optimization directives used:** Pipeline
   See /kernel4/kernel4.cpp
   See /kernel4/prj_kernel4/solution3/syn/report/kernel4_csynth.rpt
3. **Best initiation interval:** 1
4. **Optimizations:** Loop pipelining and explicit management of loop-carry dependencies.
   Direct loop pipelining has some latency reduction but it's improvable because the inter-loop data dependencies are not automatically detected and several redundant memory accesses are required. Explicit declaration of *Dependence* directive for the array variable also fails to detect data reuse so auxiliar variables and explicit loop-carry dependencies management was required. Only the final result is written in memory, not the intermediate results.
   Result: Latency reduced, initiation interval 1 reached.
5. **Resources requeriments comparison:** At every cycle we are reading two memory addresses (index[i] and array[i+1]) due to pipelining and we are also constantly using adders and multipliers. In the original solution, the loop experienced first 3 cycles in wich the main bottleneck was the memory (responsible of a higher initiation interval) and the adders and multipliers where almost not used. With the same resources we have obtained a five times faster solution.

Kernel 5.-

1. **Kernel 5 synthesis:** See /kernel5/prj_kernel5/solution1/syn/report/kernel5_csynth.rpt
2. **Optimization directives used:** Pipeline, Array partition
   See /kernel5/kernel5.cpp
   See /kernel5/prj_kernel5/solution4/syn/report/kernel5_csynth.rpt
3. **Best initiation interval:** 6

We have to face here the same issue than in the kernel 3, we have to evaluate the result of a float addition to decide if continue in the loop or finishing and this addition requires at least 4 cycles, plus another cycle and a half for evaluating the condition.

4. **Optimizations:** Loop pipelining and complete array partitioning
   We can't achieve a better initiation interval than 6. With pipelining we obtain an II of 7 but checking the schedule is easy to see that it's because of the time required for reading from memory *a[i]* and *b[i]*, we avoid this extra cycle with complete array partition.
   Result: Latency reduced about 25%
5. **Resources requeriments comparison:** Similar situation than kernel 3, memory replaced by several registers.

Kernel 6.-

1. **Kernel 6 synthesis:** See /kernel6/prj_kernel1/solution1/syn/report/kernel6_csynth.rpt
2. **Optimization directives used:** None, code not modified and no directives added
   See /kernel6/kernel6.cpp
   See /kernel6/prj_kernel6/solution1/syn/report/kernel6_csynth.rpt
3. **Best initiation interval:** 1
4. **Optimizations:** None required, loop latency is 1 cycle, no improvement is possible.
   Possible software optimization since integral square root can be computed with a less complex algorithm.
5. **Resources requeriments comparison:**

Kernel 7.-

1. **Kernel 7 synthesis:** See /kernel7/prj_kernel1/solution1/syn/report/kernel7_csynth.rpt
2. **Optimization directives used:** Pipeline
   See /kernel7/kernel7.cpp
   See /kernel7/prj_kernel7/solution2/syn/report/kernel7_csynth.rpt
3. **Best initiation interval:** 4
   We have here the same issue than in kernels 3 and 5. Float addition and comparison requires at least 4 cycles and, having an inter-loop dependence in the variable *sum*, there is no way to improve this initiation interval. This is achieved thanks to loop pipelining.
4. **Optimizations:** Loop pipelining
   With loop pipelining we obtain an initiation interval of 4 cycles, what we know is the best possible initiation interval. a and *b* arrays complete partition only reduces minimally the global latency, not the initiation interval.
   Result: Latency reduced about 60%
5. **Resources requeriments comparison:** The resources used are roughly the same but again, as the second solution is pipelined, these resources are more frequently used.

Kernel 8.-

1. **Kernel 8 synthesis:** See /kernel7/prj_kernel1/solution1/syn/report/kernel7_csynth.rpt
2. **Optimization directives used:** Array partition
   See /kernel8/kernel8.cpp
   See /kernel8/prj_kernel8/solution4/syn/report/kernel8_csynth.rpt
3. **Best initiation interval:** 2
   For offset >= 6 there is no loop-carry dependency but for 0 <= offset < 6 we have a loop carry dependency at distance 6-offset. This means that we might need the result computed in the current iteration for the next one or another one later, difficulting pipeline.
4. **Optimizations:** Complete array partition

Initiation interval one can’t be achieved because selecting the registers (array memory element after complete array partition), multiplying and writing the result takes longer than 10ns. So this combinational path has to be segmented.
We obtain thanks to complete array partition a loop iteration equal to two cycles, being the initiation interval one unachievable this means it’s an optimal implementation.
Result: Latency halved

5. **Resources requeriments comparison:** Same resources required with the exception of placing *array* in registers instead of memory for a fastest access.