Adding a new application to the FGPU. Simulation tested.

By Hector Munoz

First, the new application must be written in the opencl standard.

Step 1. Compilation of the application:

The application must be compiled using the LLVM compiler provided with the FGPU. This compilation will output a .c file that has an array of HEX instructions. This is sufficient for the implementation part. For the simulation we will need the generated ‘cram.mif’.

Step 2. Managing the cram.mif file:

```
A8000001
11010402
79000402
11000402
11010803
79000443
11000802
11010C03
79000443
11000C02
11011003
79000443
11001002
11011403
79000443
11001402
11011803
79000443
11001802
11011C03
79000443
11001C02
11012003
79000443
11002002
11012403
79000443
11002402
11012803
79000443
11002802
11012C03
79000443
11002C02
11013003
79000443
11003002
11013403
79000443
11003402
11013803
79000443
```

Figure 1. Example of the cram.mif

Next we must append this into the general cram.mif file, a comment (if not already added, with the symbols ‘--’ will help to identify where the section of our application starts).

```
2508 92000000 -- end median
2509 A8000001   --start Hello @ 2508 -- 9CC
2510 11012002
2511 79000402
2512 11000402
2513 11011403
2514 79000443
2515 11000802
2516 11013003
2517 79000443
2518 11000C02
2519 79000443
2520 11001002
2521 11013C04
2522 79000444
2523 11001402
2524 11008005
2525 79000445
2526 11001802
2527 11015C05
2528 79000445
2529 11001C02
2530 79000444
2531 11002002
2532 11014804
2533 79000444
2534 11002402
2535 79000443
2536 11002802
2537 11011003
2538 79000443
2539 11002C03
2540 11008404
2541 79000464
2542 11003003
2543 79000462
2544 92000000  --end hello
2545 A8000042  --start basic_sum @2544 -- 9F0
2546 A8000023
2547 A8000004
2548 A0000005
2549 A1000006
```
Figure 2. Adding our code to the cram.mif

Note that the file must have a maximum of 3031 lines, after our code, we must ensure that this is the case, deleting the ‘00000000’ fillers that are at the end of the cram.mif file.

Step 3. Managing the krnl_ram.mif file:

First we need to edit the lram.yml file to fit the applications configurations. Then we can use lramParser.py to convert the file to the version we need to use as the actual lram. First edit lramParser.py line 6 to have the same name as the yaml file we want to convert. The execution of the script can be done simply as: ‘python3 lramParser.py’.

We need to copy the content of our newly produced lram file into the krnl_ram.mif file. To do this make sure you replace the last kernel completely, having at the end 512 lines of the krnl_ram.mif file.

```
479 00000000
480 00000000
481 100009fb     --begin basic_sum
482 00000040
483 00000000
484 00000000
485 00000000
486 00000000
487 00000000
488 00000040
489 0000000f
490 00000000
491 00000000
492 10000040
493 00000000
494 00000000
495 00000000
496 00000000
497 10000000     --parameters
498 00000000
499 00000000
500 00000000
501 00000000
502 00000000
503 00000000
504 00000000
505 00000000
506 00000000
507 00000000
508 00000000
509 00000000
510 00000000
511 00000000
512 00000000
```

Figure 3. krnl_ram.mif file with the last kernel replaced, note the line numeration.

Step 4: Editing FGPU_simulation_pkg.vhd

First we need to add the name of our kernel to line 9, and change the CONSTANT kernel_name to our new application.

```
-------------------------------------------------------------------------------
package FGPU_simulation_pkg is
  type kernel_type is ( copy, max_half_atomic, bitonic,  fadd,  median,
floydwarshall, fir_char4, add_float, parallelSelection,  mat_mul,  fir,
xcorr,  sum_atomic,  fft_hard,   mul_float,  sobel, basic_sum);

  CONSTANT kernel_name        : kernel_type := basic_sum;
```

Figure 4. Adding the kernel's name

Also we need to add a case for our kernel:

```
            return 4;
          when floydwarshall =>
            return 5;
          when fir_char4 =>
            return 6;
          when add_float =>
            return 7;
          when parallelSelection =>
            return 8;
          when mat_mul =>
            return 9;
          when fir =>
            return 10;
          when xcorr =>
            return 11;
          when sum_atomic =>
            return 12;
          when fft_hard =>
            return 13;
          when mul_float =>
            return 14;
          when sobel =>
            return 99;
          when basic_sum =>
            return 15;
          when others =>
            assert(false) severity failure;
            return 0;
        end case;
```

Figure 5. Adding the kernel case

We must change the last kernel's number (in this case 'sobel') to a different number than one from 0 to 15 and set our kernel to '15'.

Step 5: Editing FGPU_tb.vhd

There are a few lines we must change to fit with what we configured in the lram for our application.

```
    case kernel_name is
      when floydwarshall =>
        target_param := 0;
      when copy | bitonic | add_float | parallelSelection | sum_atomic | fft_hard | sobel |
           median | max_half_atomic  =>
        target_param := 1;
      when fadd | mat_mul | fir | xcorr | mul_float | fir_char4 =>
        target_param := 2;
      when others =>
        report "undefined kernel index" severity failure;
    end case;
    s0_araddr(INTERFCE_W_ADDR_W-1 downto INTERFCE_W_ADDR_W-2) <= "00";
```

Figure 6. target_params

First we need to add our kernel's name to a case depending on the target parameters. Normally applications use 1.

Next:

```
    case kernel_name is
      when mat_mul | median | floydwarshall | sobel =>
        nDim := 2;
      when others =>
        nDim := 1;
    end case;
```

Figure 7. number of dimensions

We can leave this lines as they are unless the applications use more than one dimension.

```vhdl
    for i in 1 to 64 loop
      -- wait for 2*clk_period;
      case kernel_name is
        when mat_mul | floydwarshall | median=>
          wg_size_d0 := 8;
          wg_size_d1 := 8;
          wg_size := wg_size_d0 * wg_size_d1;
          size_d0 := wg_size_d0*i;
          size_d1 := wg_size_d1*i;
        when sobel =>
          wg_size_d0 := 4;
          wg_size_d1 := 4;
          problemSize := 16;
          size_d0 := 4;
          size_d1 := 4;
          wg_size := wg_size_d0 * wg_size_d1;
        when bitonic | fft_hard =>
          wg_size_d0 := 64;
          size_d0 := 1024*i;
          -- size_d0 := wg_size_d0*2**(i-1);
          problemSize := size_d0*2;
          wg_size := wg_size_d0;
        when others =>
          wg_size_d0 := 64;
          -- size_d0 := (i+8)*1024;
          size_d0 := 64*i;
          -- size_d0 := 128*2**(i-1);
          -- size_d0 := wg_size_d0*i;
          size_d1 := 1;
          wg_size_d1 := 1;
          wg_size := wg_size_d0;
          problemSize := REDUCE_FACTOR*size_d0;
      end case;
```

Figure 8. WG parameters.

Some parameters must be configured. If the desired configuration is not by default, we must create a new case, taking into account to configure the same variables.

```vhdl
          if kernel_name = bitonic then
            report "bitonic_kernel_finished";
          else
            report "bitonic_kernel_float finished";
          end if;
          -- }}}
        when others => -- {{{
          if (wg_size mod WF_SIZE ) = 0 then
            update_nWF_WG_sch_ram(wg_size/WF_SIZE-1);
          else
            update_nWF_WG_sch_ram(wg_size/WF_SIZE);
          end if;
          -- write_sch_ram(WG_SIZE_DX_ADDR, wg_size);
          write_sch_ram(SIZE_D0_ADDR, size_d0);
          write_sch_ram(SIZE_D1_ADDR, size_d1);
          write_sch_ram(SIZE_D2_ADDR, size_d2);
          write_WG_size_dx_sch_ram(nDim, wg_size_d0, wg_size_d1, wg_size_d2);
          update_WG_size_sch_ram(wg_size);
          write_sch_ram(N_WG_D0_ADDR, max(size_d0/wg_size_d0-1, 0));
          write_sch_ram(N_WG_D1_ADDR, max(size_d1/wg_size_d1-1, 0));
          if kernel_name = max_half_atomic or kernel_name = sum_atomic then
            write_param(REDUCE_FACTOR, 3);
          end if;
          size_0 <= size_d0;
          size_1 <= size_d1;
          set_write_cache;
          start_kernel(1);
          -- clear_initialize;
          wait_to_finish; -- }}}
      end case;
    end loop;
    report "END of simulation" severity failure;
```

Figure 9. Writing into RAM

In line 908 a case starts for writing the values to the LRAM. Normally, a new application would use the default configuration found in Figure 9 (when others).

Step 6: Editing global_mem.vhd

```vhdl
    if new_kernel = '1' then
      if kernel_name = mat_mul or kernel_name = xcorr  then
        gmem <= init_me_with_modulu(C_MEM_SIZE/2, FILL_MODULO);
      elsif kernel_name = fadd or kernel_name = add_float or kernel_name = mul_float
        gmem <= init_mem_rand(C_MEM_SIZE/2, 32);
      elsif kernel_name = floydwarshall then
        gmem <= init_mem_floydwarshall(C_MEM_SIZE/2);
      elsif kernel_name = fft_hard then
        gmem <= init_mem_fft(size_0);
      elsif kernel_name = fir_char4 then
        gmem <= init_mem(C_MEM_SIZE/2, 8);
      elsif kernel_name = parallelSelection then
        gmem <= init_mem_float(C_MEM_SIZE/2);
      -- elsif kernel_name = ludecompose then
      --   gmem <= init_mem_rand(C_MEM_SIZE/2, 32);
      --   -- gmem(0)(DATA_W-1 downto 0) <= to_slv(to_float(121));
      --   -- gmem(0)(2*DATA_W-1 downto DATA_W) <= to_slv(to_float(68));
      --   -- gmem(1)(DATA_W-1 downto 0) <= to_slv(to_float(30));
      --   -- gmem(1)(2*DATA_W-1 downto DATA_W) <= to_slv(to_float(73));
      --   -- gmem(2)(DATA_W-1 downto 0) <= to_slv(to_float(109));
      --   -- gmem(2)(2*DATA_W-1 downto DATA_W) <= to_slv(to_float(94));
      --   -- gmem(3)(DATA_W-1 downto 0) <= to_slv(to_float(62));
      --   -- gmem(3)(2*DATA_W-1 downto DATA_W) <= to_slv(to_float(31));
      --   -- gmem(4)(DATA_W-1 downto 0) <= to_slv(to_float(113));
      --   -- gmem(4)(2*DATA_W-1 downto DATA_W) <= to_slv(to_float(5));
      --   -- gmem(5)(DATA_W-1 downto 0) <= to_slv(to_float(27));
      --   -- gmem(5)(2*DATA_W-1 downto DATA_W) <= to_slv(to_float(106));
      --   -- gmem(6)(DATA_W-1 downto 0) <= to_slv(to_float(33));
      --   -- gmem(6)(2*DATA_W-1 downto DATA_W) <= to_slv(to_float(6));
      --   -- gmem(7)(DATA_W-1 downto 0) <= to_slv(to_float(86));
      --   -- gmem(7)(2*DATA_W-1 downto DATA_W) <= to_slv(to_float(92));
      else
        gmem <= init_mem(C_MEM_SIZE/2, 32);
      end if;
    end if;
  end if;
```

Figure 10. Memory initialization

In Figure 10 we configure how the memory gets initialized for our application (with which values). They can be random or not. We can add our kernel to the case that fits our needs. If leaved alone, the memory will be initialized with the default case (ascending order).

```vhdl
  if wvalid(i) = '1' and wready(i) = '1' then
    wr_addr_int := to_integer(unsigned(wr_addr_offset(i)));
    -- assert wr_addr_int /= 16#1b9b8# and wr_addr_int /= 16#1b9b9# and wr_addr_int /= 16#1b9ba# and wr_addr_i
    -- write(output, "0x" & to_hstring(to_signed(word_addr, 32)) & LF);
    if kernel_name = bitonic or kernel_name = fft_hard or kernel_name = floydwarshall then
      word_addr := wr_addr_int*2; -- index of first parameter value
    else
      word_addr := wr_addr_int*2-(offset+target_offset_addr)/4; -- index of first parameter value
    end if;
    second_word_addr := word_addr + 64*1024; -- index of the second parameter
```

Figure 11. Index of parameters.

The index of the first parameter in the memory can be configured in the lines shown in Figure 11. If left alone, the default index will take place.

```
        case kernel_name is
          when copy  =>
            for k in 0 to 1  loop
              must_data((k+1)*DATA_W-1 downto k*DATA_W) := std_logic_vector(to_unsigned(word_addr+k, DATA_W));
            end loop;
          when parallelSelection  =>
            for k in 0 to 1  loop
              -- must_data((k+1)*DATA_W-1 downto k*DATA_W) := std_logic_vector(to_unsigned(word_addr+k, DATA_W));
              must_data((k+1)*DATA_W-1 downto k*DATA_W) := std_logic_vector(to_float(word_addr+k));
            end loop;
          when max_half_atomic =>
            must_data(DATA_W-1 downto 0) := tmp_gmem(0);
          when sobel =>
          when bitonic | fft_hard | median =>
            must_data(DATA_W-1 downto 0) := tmp_gmem(word_addr);
            -- report integer'image(word_addr);
            -- report integer'image(to_integer(unsigned(tmp_gmem(word_addr))));
            -- report integer'image(to_integer(unsigned(tmp_gmem(word_addr+1))));
            must_data(2*DATA_W-1 downto DATA_W) := tmp_gmem(word_addr+1);
```

Figure 12. Must data configuration

The must_data is a variable that gets filled with how the memory should look like at the end of the kernel's execution. Here we need to add our kernel's name (there is no default version) to an existing case, or create our own case.

The FGPU simulation will use these configurations to compare with floating point values, word sized values, or byte sized values, to give an example.

```
1222              elsif kernel_name /= sum_atomic then
1223                if  wstrb(i)((k+1)*DATA_W/8-1 downto k*DATA_W) = X"F" and
1224                    must_data((k+1)*DATA_W-1 downto k*DATA_W) /= wdata(i)((k+1)*DATA_W-1 downto k*DATA_W) then
1225                  write(output, "wdata word " & integer'image(k) & " on AXI " & integer'image(i) & " is " &
1226                       integer'image(to_integer(unsigned(wdata(i)((k+1)*DATA_W-1 downto k*DATA_W)))) &
1227                       " (should be " & integer'image(to_integer(unsigned(must_data((k+1)*DATA_W-1 downto k*DATA_W)))) & ")
1228                       integer'image(word_addr+k) & LF);
1229                  assert false severity failure;
1230                end if;
1231              end if;
1232              if wstrb(i)(k*DATA_W/8) = '1' then
1233                if written_addrs(word_addr+k) = '0' then
1234                  written_count_tmp := written_count_tmp + 1;
1235                else
1236                  -- report "double write";
1237                end if;
```

Figure 13. Assertions for the must data.

The default configuration for this part can be left alone but it might be necessary to comment out lines 1223 - 1230 if the assertion fails in the simulation.

```
        elsif  kernel_name = fft_hard then
          fft_round;
          stageIndx := stageIndx + 1;
        elsif kernel_name = sum_atomic or kernel_name = max_half_atomic then
          assert  must_data(DATA_W-1 downto 0) = gmem(65536)(DATA_W-1 downto 0)
          report "result is " & integer'image(to_integer(unsigned(gmem(65536)(DATA_W-1 downto 0))))& " (must be " &
                  integer'image(to_integer(unsigned(must_data(DATA_W-1 downto 0)))) & ")"
                  severity failure;
          -- report "wdata word " & integer'image(k) & " on AXI " & integer'image(i) & " is " &
          --         integer'image(to_integer(unsigned(wdata(i)((k+1)*DATA_W-1 downto k*DATA_W)))) &
          --         " (should be " & integer'image(to_integer(unsigned(must_data((k+1)*DATA_W-1 downto k*DATA_W)))) &
          --         integer'image(word_addr+k) severity failure;
          -- report integer'image(to_integer(unsigned(must_data(DATA_W-1 downto 0))));
          check_written_count(1);
        else
          if COMP_TYPE = 0 then -- byte mode
            check_written_count(size_0*size_1*4);
          elsif kernel_name = median then
            check_written_count(size_0*size_1-2*(size_0-1)-2*(size_1-1)); -- no write for edge pixels
          else
            check_written_count(size_0*size_1);
          end if;
        end if;
      end if;
```

Figure 14. Checking the values.

Finally, the default configuration can be left alone to be executed, the simulation will compare the results to the expected output.