Accelerating DICe on FPGA

Student: Atiyehsadat Panahi

Advisor: David Andrews

1. What is done

1.1. Introduction

As it had been mentioned in the previous report, the main time consuming part of the DICe are the objective functions. The objective functions do the optimization part of the correlations. They iterate through the whole deformed frame to find the equivalent regions of the reference frame in the deformed frame and then find the displacement. It is done using two methods of fast and robust. The fast method is typically an image gradient based method and the robust method is usually simplex based. In these two functions that are called for each subset of the image frames, the intensity values of the whole pixels are compared between the reference and the deformed images. The fast method has a for loop with 25 iterations and the robust method loops through the image 100 times. We assume that there are around 15k frames (as in the sample input) and each frame has 5 subsets with 40*40 pixels. It means that the total number of iterations will be 15k * 5 * 40 *40 * 25 = 3e9 for the fast method and 12e9 for the robust method. That's why we will start from these functions to accelerate them on the FPGA to have an optimization in the execution time.

1.2. Objective functions

We have started from the fast method and rewrote this function in Verilog. Figure 1 and Figure 2 are the fast method implementation in C++ and also in Verilog. This function is implemented using an FSM-based switch case in an always block to handle the loops with an unfixed loop iteration value. Shon in Figure 3 is the simulation result of this function for a simple 4 * 4 subset. As the waveforms determine, the Correlation_Done is the output of this function that determines the status of the correlation that is done or not.

```
Status_Flag
Objective_ZNSSD::computeUpdateFast(Teuchos::RCP<Local_Shape_Function> shape_function,
  int_t & num_iterations){
  TEUCHOS_TEST_FOR_EXCEPTION(!subset_->has_gradients(),std::runtime_error,"Error, image gradients have not been computed but are needed here.");
  // TODO catch the case where the initial gamma is good enough (possibly do this at the image level, not subset?):
  int_t N = shape_function->num_params(); // one degree of freedom for each shape function parameter
  assert(N>=2);
  scalar_t tolerance = schema_->fast_solver_tolerance();
  const int_t max_solve_its = schema_->max_solver_iterations_fast();
  int *IPIV = new int[N+1];
  int LWORK = N*N;
  int INFO = 0;
  // using type double here because LAPACK doesn't support float.
  double *WORK = new double[LWORK];
  Teuchos::LAPACK<int_t,double> lapack;

  // Initialize storage:
  Teuchos::SerialDenseMatrix<int_t,double> H(N,N, true);
  Teuchos::ArrayRCP<double> q(N,0.0);
  std::vector<scalar_t> residuals(N,0.0);
  std::vector<scalar_t> def_old(N,0.0);     // save off the previous value to test for convergence
  std::vector<scalar_t> def_update(N,0.0); // save off the previous value to test for convergence

  // note this creates a pointer to the array so
  // the values are updated each frame if compute_grad_def_images is on
  Teuchos::ArrayRCP<scalar_t> gradGx = subset_->grad_x_array();
  Teuchos::ArrayRCP<scalar_t> gradGy = subset_->grad_y_array();
  const scalar_t cx = subset_->centroid_x();
  const scalar_t cy = subset_->centroid_y();
  const scalar_t meanF = subset_->mean(REF_INTENSITIES);

  scalar_t old_u=0.0,old_v=0.0,old_t=0.0;
  shape_function->map_to_u_v_theta(cx,cy,old_u,old_v,old_t);
  DEBUG_MSG(std::setw(5) << "Iter" <<
    std::setw(12) << " u" <<
    std::setw(12) << " du" <<
    std::setw(12) << " v" <<
    std::setw(12) << " dv" <<
    std::setw(12) << " t" <<
    std::setw(12) << " dt");

  int_t solve_it = 0;
  for(;solve_it<=max_solve_its;++solve_it){
    num_iterations = solve_it;

    // update the deformed image with the new deformation:
    try{
      subset_->initialize(schema_->def_img(subset_->sub_image_id()),DEF_INTENSITIES,shape_function,schema_->interpolation_method());
      //#ifdef DICE_DEBUG_MSG
      //      std::stringstream fileName;
      //      fileName << "defSubset_" << correlation_point_global_id_ << "_" << solve_it;
Teuchos::ArrayRCP<scalar_t> gradGy = subset_->grad_y_array();
const scalar_t cx = subset_->centroid_x();
const scalar_t cy = subset_->centroid_y();
const scalar_t meanF = subset_->mean(REF_INTENSITIES);

scalar_t old_u=0.0,old_v=0.0,old_t=0.0;
shape_function->map_to_u_v_theta(cx,cy,old_u,old_v,old_t);
DEBUG_MSG(std::setw(5) << "Iter" <<
  std::setw(12) << " u" <<
  std::setw(12) << " du" <<
  std::setw(12) << " v" <<
  std::setw(12) << " dv" <<
  std::setw(12) << " t" <<
  std::setw(12) << " dt");

int_t solve_it = 0;
for(;solve_it<=max_solve_its;++solve_it){
  num_iterations = solve_it;

  // update the deformed image with the new deformation:
  try{
    subset_->initialize(schema_->def_img(subset_->sub_image_id()),DEF_INTENSITIES,shape_function,schema_->interpolation_method());
    //#ifdef DICE_DEBUG_MSG
    //      std::stringstream fileName;
    //      fileName << "defSubset_" << correlation_point_global_id_ << "_" << solve_it;
  catch (std::logic_error & err) {
    return SUBSET_CONSTRUCTION_FAILED;
  }
  // compute the mean value of the subsets:
  const scalar_t meanG = subset_->mean(DEF_INTENSITIES);
  // the gradients are taken from the def images rather than the ref
  const bool use_ref_grads = schema_->def_img()->has_gradients() ? false : true;

  scalar_t GmF = 0.0;
  for(int_t index=0;index<subset_->num_pixels();++index){
    if(subset_->is_deactivated_this_step(index)||!subset_->is_active(index)) continue;
    GmF = (subset_->def_intensities(index) - meanG) - (subset_->ref_intensities(index) - meanF);
    for(int_t i=0;i<N;++i)
      residuals[i] = 0.0;
    shape_function->residuals(subset_->x(index),subset_->y(index),cx,cy,gradGx[index],gradGy[index],residuals,use_ref_grads);
    for(int_t i=0;i<N;++i){
      q[i] += GmF*residuals[i];
      for(int_t j=0;j<N;++j)
        H(i,j) += residuals[i]*residuals[j];
    }
  }

  if(schema_->use_objective_regularization()){ // TODO test for affine shape functions too
    // add the penalty terms
    const scalar_t alpha = schema_->levenberg_marquardt_regularization_factor();
    H(0,0) += alpha;
    H(1,1) += alpha;
  }

  // compute the norm of H prior to taking the inverse:
  // Note: for this to work, the shape functions must always have their displacement degrees of freedom as the
  // first two parameters (assert that N>=2 above)
  const scalar_t det_h = H(0,0)*H(1,1) - H(1,0)*H(0,1);
  const scalar_t norm_H = std::sqrt(H(0,0)*H(0,0) + H(0,1)*H(0,1) + H(1,0)*H(1,0) + H(1,1)*H(1,1));
  scalar_t cond_2x2 = -1.0;
  if(det_h !=0.0){
    const scalar_t norm_Hi = std::sqrt((1.0/(det_h*det_h))*(H(0,0)*H(0,0) + H(0,1)*H(0,1) + H(1,0)*H(1,0) + H(1,1)*H(1,1)));
    cond_2x2 = norm_H * norm_Hi;
  }

  // clear temp storage
  for(int_t i=0;i<LWORK;++i) WORK[i] = 0.0;
  for(int_t i=0;i<N+1;++i) {IPIV[i] = 0;}
  try
  {
    lapack.GETRF(N,N,H.values(),N,IPIV,&INFO);
    if(correlation_point_global_id_>=0)
      schema_->global_field_value(correlation_point_global_id_,CONDITION_NUMBER_FS) = cond_2x2;
    if(cond_2x2 > 1.0E12) return HESSIAN_SINGULAR;
  }
  catch(std::exception &e){
    DEBUG_MSG( e.what() << '\n');
    return LINEAR_SOLVE_FAILED;
  }
  // save off last step
  for(int_t i=0;i<N;++i)
    def_old[i] = (*shape_function)(i);
  for(int_t i=0;i<N;++i)
    def_update[i] = 0.0;
  for(int_t i=0;i<N;++i)
    for(int_t j=0;j<N;++j)
      def_update[i] += H(i,j)*(-1.0)*q[j];
  shape_function->update(def_update);

  scalar_t guess_u = 0.0,guess_v=0.0,guess_t=0.0;
  shape_function->map_to_u_v_theta(cx,cy,guess_u,guess_v,guess_t);
  std::ios  state(NULL);
  state.copyfmt(std::cout);
  DEBUG_MSG(std::setw(5) << solve_it <<
    std::setw(12) << std::scientific << std::setprecision(4) << guess_u <<
    std::setw(12) << guess_u - old_u <<
    std::setw(12) << guess_v <<
    std::setw(12) << guess_v - old_v <<
    std::setw(12) << guess_t <<
    std::setw(12) << guess_t - old_t);
  std::cout.copyfmt(state);
  shape_function->map_to_u_v_theta(cx,cy,old_u,old_v,old_t);

  const bool converged = shape_function->test_for_convergence(def_old,tolerance);
  if(converged){
    DEBUG_MSG("Subset " << correlation_point_global_id_ << " ** CONVERGED SOLUTION ");
    shape_function->print_parameters();
    computeUncertaintyFields(shape_function);
    break;
  }

  // zero out the storage
  H.putScalar(0.0);
  for(int_t i=0;i<N;++i)
    q[i] = 0.0;
} // end solve iteration loop

// clean up storage for lapack:
delete [] WORK;
delete [] IPIV;

if(solve_it>max_solve_its){
  return MAX_ITERATIONS_REACHED;
}
else return CORRELATION_SUCCESSFUL;
}
```

```
Status_Flag
Objective_ZNSSD::computeUpdateFast(Teuchos::RCP<Local_Shape_Function> shape_function,
  int_t & num_iterations){
  TEUCHOS_TEST_FOR_EXCEPTION(!subset_->has_gradients(),std::runtime_error,"Error, image gradients have not been computed but are needed here.");
  // TODO catch the case where the initial gamma is good enough (possibly do this at the image level, not subset?):
  int_t N = shape_function->num_params(); // one degree of freedom for each shape function parameter
  assert(N>=2);
  scalar_t tolerance = schema_->fast_solver_tolerance();
  const int_t max_solve_its = schema_->max_solver_iterations_fast();
  int *IPIV = new int[N+1];
  int LWORK = N*N;
  int INFO = 0;
  // using type double here because LAPACK doesn't support float.
  double *WORK = new double[LWORK];
  Teuchos::LAPACK<int_t,double> lapack;

  // Initialize storage:
  Teuchos::SerialDenseMatrix<int_t,double> H(N,N, true);
  Teuchos::ArrayRCP<double> q(N,0.0);
  std::vector<scalar_t> residuals(N,0.0);
  std::vector<scalar_t> def_old(N,0.0);     // save off the previous value to test for convergence
  std::vector<scalar_t> def_update(N,0.0); // save off the previous value to test for convergence

  // note this creates a pointer to the array so
  // the values are updated each frame if compute_grad_def_images is on
  Teuchos::ArrayRCP<scalar_t> gradGx = subset_->grad_x_array();
  Teuchos::ArrayRCP<scalar_t> gradGy = subset_->grad_y_array();
  const scalar_t cx = subset_->centroid_x();
  const scalar_t cy = subset_->centroid_y();
  const scalar_t meanF = subset_->mean(REF_INTENSITIES);

  scalar_t old_u=0.0,old_v=0.0,old_t=0.0;
  shape_function->map_to_u_v_theta(cx,cy,old_u,old_v,old_t);
  DEBUG_MSG(std::setw(5) << "Iter" <<
    std::setw(12) << " u" <<
    std::setw(12) << " du" <<
    std::setw(12) << " v" <<
    std::setw(12) << " dv" <<
    std::setw(12) << " t" <<
    std::setw(12) << " dt");

  int_t solve_it = 0;
  for(;solve_it<=max_solve_its;++solve_it){
    num_iterations = solve_it;

    // update the deformed image with the new deformation:
    try{
      subset_->initialize(schema_->def_img(subset_->sub_image_id()),DEF_INTENSITIES,shape_function,schema_->interpolation_method());
      //#ifdef DICE_DEBUG_MSG
      //      std::stringstream fileName;
      //      fileName << "defSubset_" << correlation_point_global_id_ << "_" << solve_it;
  Teuchos::ArrayRCP<scalar_t> gradGy = subset_->grad_y_array();
  const scalar_t cx = subset_->centroid_x();
  const scalar_t cy = subset_->centroid_y();
  const scalar_t meanF = subset_->mean(REF_INTENSITIES);

  scalar_t old_u=0.0,old_v=0.0,old_t=0.0;
  shape_function->map_to_u_v_theta(cx,cy,old_u,old_v,old_t);
  DEBUG_MSG(std::setw(5) << "Iter" <<
    std::setw(12) << " u" <<
    std::setw(12) << " du" <<
    std::setw(12) << " v" <<
    std::setw(12) << " dv" <<
    std::setw(12) << " t" <<
    std::setw(12) << " dt");

  int_t solve_it = 0;
  for(;solve_it<=max_solve_its;++solve_it){
    num_iterations = solve_it;

    // update the deformed image with the new deformation:
    try{
      subset_->initialize(schema_->def_img(subset_->sub_image_id()),DEF_INTENSITIES,shape_function,schema_->interpolation_method());
      //#ifdef DICE_DEBUG_MSG
      //      std::stringstream fileName;
      //      fileName << "defSubset_" << correlation_point_global_id_ << "_" << solve_it;
    catch (std::logic_error & err) {
      return SUBSET_CONSTRUCTION_FAILED;
    }
    // compute the mean value of the subsets:
    const scalar_t meanG = subset_->mean(DEF_INTENSITIES);
    // the gradients are taken from the def images rather than the ref
    const bool use_ref_grads = schema_->def_img()->has_gradients() ? false : true;

    scalar_t GmF = 0.0;
    for(int_t index=0;index<subset_->num_pixels();++index){
      if(subset_->is_deactivated_this_step(index)||!subset_->is_active(index)) continue;
      GmF = (subset_->def_intensities(index) - meanG) - (subset_->ref_intensities(index) - meanF);
      for(int_t i=0;i<N;++i)
        residuals[i] = 0.0;
      shape_function->residuals(subset_->x(index),subset_->y(index),cx,cy,gradGx[index],gradGy[index],residuals,use_ref_grads);
      for(int_t i=0;i<N;++i){
        q[i] += GmF*residuals[i];
        for(int_t j=0;j<N;++j)
          H(i,j) += residuals[i]*residuals[j];
      }
    }

    if(schema_->use_objective_regularization()){ // TODO test for affine shape functions too
      // add the penalty terms
      const scalar_t alpha = schema_->levenberg_marquardt_regularization_factor();
      H(0,0) += alpha;
      H(1,1) += alpha;
    }

    // compute the norm of H prior to taking the inverse:
    // Note: for this to work, the shape functions must always have their displacement degrees of freedom as the
    // first two parameters (assert that N>=2 above)
    const scalar_t det_h = H(0,0)*H(1,1) - H(1,0)*H(0,1);
    const scalar_t norm_H = std::sqrt(H(0,0)*H(0,0) + H(0,1)*H(0,1) + H(1,0)*H(1,0) + H(1,1)*H(1,1));
    scalar_t cond_2x2 = -1.0;
    if(det_h !=0.0){
      const scalar_t norm_Hi = std::sqrt((1.0/(det_h*det_h))*(H(0,0)*H(0,0) + H(0,1)*H(0,1) + H(1,0)*H(1,0) + H(1,1)*H(1,1)));
      cond_2x2 = norm_H * norm_Hi;
    }

    // clear temp storage
    for(int_t i=0;i<LWORK;++i) WORK[i] = 0.0;
    for(int_t i=0;i<N+1;++i) {IPIV[i] = 0;}
    try
    {
      lapack.GETRF(N,N,H.values(),N,IPIV,&INFO);
      if(correlation_point_global_id_>=0)
        schema_->global_field_value(correlation_point_global_id_,CONDITION_NUMBER_FS) = cond_2x2;
      if(cond_2x2 > 1.0E12) return HESSIAN_SINGULAR;
    }
    catch(std::exception &e){
      DEBUG_MSG( e.what() << '\n');
      return LINEAR_SOLVE_FAILED;
    }
    // save off last step
    for(int_t i=0;i<N;++i)
      def_old[i] = (*shape_function)(i);
    for(int_t i=0;i<N;++i)
      def_update[i] = 0.0;
    for(int_t i=0;i<N;++i)
      for(int_t j=0;j<N;++j)
        def_update[i] += H(i,j)*(-1.0)*q[j];
    shape_function->update(def_update);

    scalar_t guess_u = 0.0,guess_v=0.0,guess_t=0.0;
    shape_function->map_to_u_v_theta(cx,cy,guess_u,guess_v,guess_t);
    std::ios  state(NULL);
    state.copyfmt(std::cout);
    DEBUG_MSG(std::setw(5) << solve_it <<
      std::setw(12) << std::scientific << std::setprecision(4) << guess_u <<
      std::setw(12) << guess_u - old_u <<
      std::setw(12) << guess_v <<
      std::setw(12) << guess_v - old_v <<
      std::setw(12) << guess_t <<
      std::setw(12) << guess_t - old_t);
    std::cout.copyfmt(state);
    shape_function->map_to_u_v_theta(cx,cy,old_u,old_v,old_t);

    const bool converged = shape_function->test_for_convergence(def_old,tolerance);
    if(converged){
      DEBUG_MSG("Subset " << correlation_point_global_id_ << " ** CONVERGED SOLUTION ");
      shape_function->print_parameters();
      computeUncertaintyFields(shape_function);
      break;
    }

    // zero out the storage
    H.putScalar(0.0);
    for(int_t i=0;i<N;++i)
      q[i] = 0.0;
  } // end solve iteration loop

  // clean up storage for lapack:
  delete [] WORK;
  delete [] IPIV;

  if(solve_it>max_solve_its){
    return MAX_ITERATIONS_REACHED;
  }
  else return CORRELATION_SUCCESSFUL;
}
```

Figure 1-C++ Code

Figure 2-Verilog Code

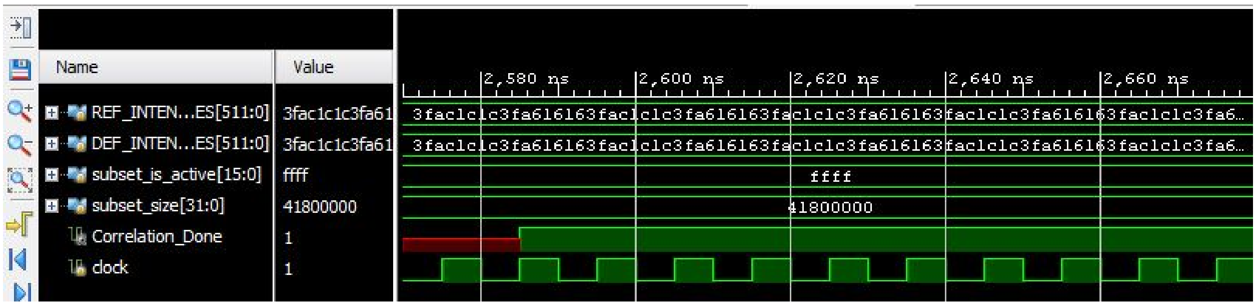

Figure 3- Simulation Results.

To synthesize the design, we have to do some initializations at the upper level modules. The problem is that we have an object oriented C++ code that it should be recoded in Verilog which is a low level language that even does not support 2D arrays as the input port or using the dynamic arrays and loops. We are working on this part now. But to be sure that the function works fine in synthesizing we have tested it with a simple test bench that the data have been initialized by hand not in a hierarchy structure as is in C++ source codes. The left most led in the Vitex 7 FPGA represents the Correlation_Done output signal (like the simulation waveforms).

Figure 4- Synthesize Results.

1.3. Arithmetic functions

Arithmetic operations of the DICe is in ieee754 single precision format. To have them in Verilog, we could use the floating point IP core of the Vivado that implements the basic arithmetic functions or use the present libraries but, most of them are sequential circuits that cannot be implemented as a function and we have to have functions because of their features compared to the modules. Therefore, we used combinational arithmetic cells like adder/subtractor, multiplier, divider, sqrt, sin, cos, asin and acos. For the trigonometric functions the Taylor series has been used to have a linear approximation of these functions. They have been worked for both the simulation and synthesizing for a simple test of reading two inputs from the BRAM, doing the arithmetic functions and having the results on the FPGA at a 70 MHz frequency.

2. Next steps

1. How to have an object-oriented-based code in Verilog!
2. Coding the upper level functions, initializing the needed variables and then synthesizing the design.
3. Analyzing the execution time of the fast method (including the time of reading the data from BRAM and the execution time of the other functions)
4. Coding the robust method.
5. Test the design for larger input size data.