TBB: parallel_pipeline

OBJECTIVES

- Learn about software pipelining.
- Implement pipelining via Threading Building Blocks (TBB) Library in C++: *parallel_pipeline*

USEFUL INFORMATION

- Refer to the Tutorial: Embedded Intel for the source files used in this Tutorial.
- Refer to the board website or the Tutorial: Embedded Intel for User Manuals and Guides for the Terasic DE2i-150 Board.
- Board Setup: Connect the monitor (VGA or HDMI) as well as the keyboard and mouse.
  - Refer to the *DE2i-150 Quick Start Guide* (page 2) for a useful illustration.

ACTIVITIES

FIRST ACTIVITY: MODULUS

- For two n-element vectors $\vec{a}$ and $\vec{b}$, we want to calculate the modulus of each pair of elements and place the results on $\vec{c}$.

$$c(i) = \sqrt{a(i)^2 + b(i)^2}$$

- Example: `a = [3 4 1 10 6 1.5 2.5 5 8 7]  b = [4 3 1 2 8 1.5 6 12 15 24]`
  - ✓ Result: `c = [5 5 1.4142 10.198 10 2.1213 6.5 13 17 25[`

Pipelined implementation

- Though there are many ways to implement this in a parallel fashion (including map), here we use a serial-parallel-serial pipeline for illustration purposes.
- Using *parallel_pipeline*, we define 3 filters and the associated functors. Each functor is defined as a class.
  - ✓ Caution: Because the body object provided to the filters of the *parallel_pipeline* might be copied, its `operator()` should not modify the body. Otherwise the modification might or might not become visible to the thread that invoked *parallel_pipeline*, depending upon whether `operator()` is acting on the original or a copy. As a reminder of this nuance, *parallel_pipeline* requires that the body object's `operator()` be declared `const`.

- We invoke *parallel_pipeline* within a function. Input to each stage ≡ input parameter to the `operator()` inside each functor. When invoking the functor in `make_filter` we can feed other parameters (via parameterized constructor), but this is not data that flows through the pipeline. The data type of the input/output of each stage is specified in `make_filter <X,Y>`.

```
void RunPipeline (int ntoken, int n, float *a, float *b, float *c) {
   parallel_pipeline(ntoken,    // 'filter_mode' instead on 'filter' in latest tbb
          make_filter< void, MyMod>(filter_mode::serial_in_order, my_in(a,b,n) )
      &   make_filter<MyMod, float>(filter_mode::parallel,        my_transf() )
      &   make_filter<float,  void>(filter_mode::serial_in_order, my_out(c) ) );
}
```

  - ✓ First Stage (defined by functor `my_in`): Syntax-wise, this stage has no input (only a `flow_control` object is passed to its functor). Input parameters to functor: arrays `a` and `b`, and variable `n`. The stage outputs a `MyMod` value (pair of floats).

```
class my_in {
   float *a;
   float *b;
   int n;
   mutable int i;

public:
   my_in (float *ap, float *bp, int np): a(ap), b(bp), n(np), i(0) {} // a=ap, b=bp, n=np, i=0
   MyMod operator () (flow_control& fc) const {
      MyMod t;
      const MyMod ret_val = {.av = 0 , .bv = 0 };
      if (i < n) { t.av = *(a+i); t.bv = *(b+i); i++; return t; }
      else  { fc.stop(); return ret_val; } // 'return ret_val' (NULL values) required in last tbb
      }
};
```

    - `MyMod` class:

```
class MyMod {
public:
  float av;
  float bv;
};
```

- Note that we use the index `i` (that changes during the execution of the pipeline) to determine if we reached the end of the arrays. The way they are defined, `a` and `b` do not change their values.

✓ Second stage (defined by functor `my_transf`): It performs the operation. As it is a relatively complex operation, it is advantageous to use a parallel stage so that elements can be processed concurrently.

```
class my_transf {
public:
  float operator() (MyMod input) const { // 'input': implicitly provided by RunPipeline
     float result;

     result = sqrt ( (input.av)*(input.av) + (input.bv)*(input.bv));
     return result;
  }
};
```

✓ Third Stage (defined by functor `my_out`): Syntax-wise, this stage has not outputs. The input parameter to the functor is the array `c` and this stage places elements on this array `c`.

```
class my_out {
  mutable int j; // so it can be modified by a 'const' operator()

public:
  float *ci;
  my_out (float *cp): ci(cp), j(0) {} // initialize constructor. ci = cp, j = 0

  void operator () (float result) const {
    *(ci+j) = result;
    j++;
  }
};
```

- This is an example of execution of the pipeline:

```
int main () {
  int n = 10; // Feel free to modify n
  int ntoken = 16;
  float a[10], b[10], c[10];
  int i;

  for (i=0; i < n; i++) {
    a[i] = sin(i * 3.1416/n);
    b[i] = tan(i * 3.1416/n); }

  RunPipeline (ntoken, n, a, b, c);

  cout << "Result:\n";
  for (i = 0; i < n; i++) { cout << c[i] << "\n"; }
  return 0;
}
```

- Fig. 1 depicts the pipeline and the operations at each stage.

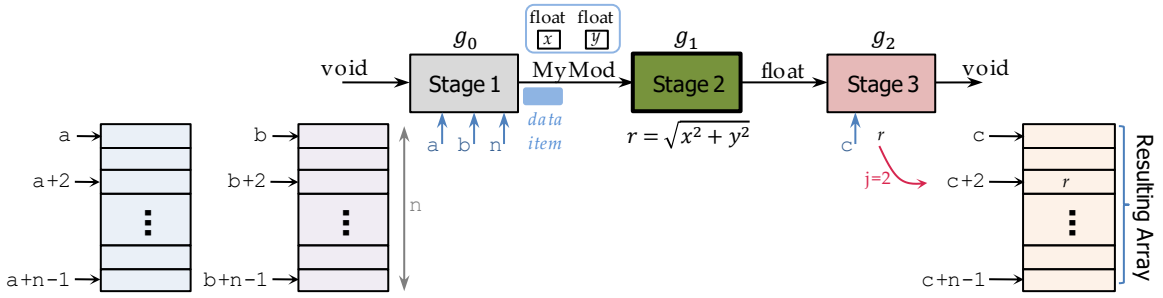

Figure 1. Serial-parallel-serial pipeline. Data provided as input parameters a and b. Stage 1 feeds input items into the pipeline. Parallel Stage 2 performs the modulus of the MyMod item (items can be processed in parallel). Stage 3 places the result on an output array.

- Application files: `pip_mod.cpp`
  ✓ We use `using namespace tbb` to avoid including the prefix `tbb` before each identifier used by the `tbb` library.

- Compile this application: `g++ pip_mod.cpp -ltbb -o pip_mod ↵`
- Execute this application: Feel free to modify `n` in order to compute larger sequences
  `./pip_mod ↵`

SECOND ACTIVITY: SUM OF SQUARED VALUES

- For an n-element vector $\vec{p}$, we want to calculate the sum of squared elements and place the results on a variable x.

$$x = \sum_{i=first}^{last-1} p(i)^2$$

- In the code, the array $\vec{p}$ is defined by pointer `first`. The last element is given by `last-1`. Thus, the pointers to every element of the array $\vec{p}$ are given by `[first, last)`.

Pipelined implementation

- Though this is a reduction problem (suited *parallel_reduce*), we use a serial-parallel-serial pipeline for illustration purposes.
- Using *parallel_pipeline*, we define 3 filters and the associated functors. Each functor is defined in a compact λ expression.
- Here, we show a function (`SumSquare`) where *parallel_pipeline* is invoked. Note that we define the functors g_0, g_1, g_2 using compact lambda expressions (no need for class definition). This requires the `-std=c+11` modifier at compilation.

```
float SumSquare( float* first, float* last ) {
   float sum = 0;
   parallel_pipeline (16, // ntoken = 16
                      make_filter<void,float*>(filter_mode::serial_in_order,
                                               [&](flow_control& fc)-> float* { //functor g0: λ exprsn
                                                      if( first < last ) {
                                                         return first++;
                                                      } else {
                                                         fc.stop();
                                                         return NULL;
                                                      }
                                               } ) &
                      make_filter<float*,float>(filter_mode::parallel,
                                                [](float* p) { return (*p)*(*p); } ) &
                      make_filter<float,void>  (filter_mode::serial_in_order,
                                                [&](float x) { sum += x; })  );
   return sum;
}
```

✓ In these compact lambda expressions, the input to each stage is specified before the statements in `{…}`, whereas the outputs are specified via the `return` keyword. We can also feed input parameters to the functors associated with stage just by using the available variables (these parameters do not flow through the pipeline.

- This is an example of execution of the pipeline:

```
int main( ) {
   int i;
   float fi[101], *fo, ff;

   for (i = 0; i < 100; i++) fi[i] = i;
   fo = &fi[100]; // fi[100] will not be considered

   ff = SumSquare (fi, fo); // first=fi, last = fo
   cout << ff << "\n"; // sum of the squares of 0 to 99: 328350
   return 0;
}
```

- Fig. 2 depicts the pipeline and the operations at each stage.

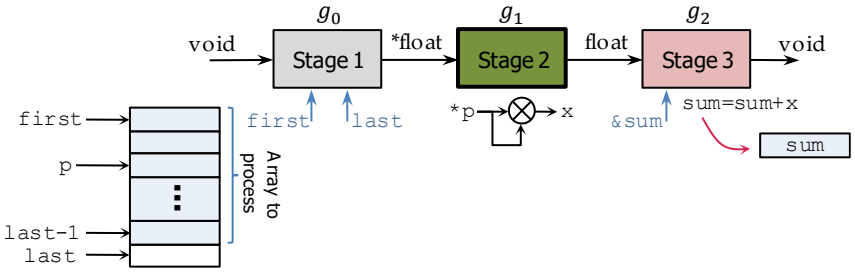

Figure 2. Serial-parallel-serial pipeline. Data is provided as input parameters (pointers *first* and *last*). Stage 1 feeds input items into the pipeline. Parallel Stage 2 performs the squaring of an item (items can be processed in parallel). Stage 3 accumulates the result one item at a time.

- Application files: `pip_sumsq.cpp`
- Compile this application: `g++ -std=c++11 pip_sumsq.cpp -ltbb -o pip_sumsq ↵`
- Execute this application: `./pip_sumsq ↵`

THIRD ACTIVITY: PROCESSING VECTORS

- We want to process NV sets of two n-element vectors, where a set of two vectors constitutes an item fed to the pipeline.
- For each set (vectors $\vec{x}$ and $\vec{y}$), we compute a resulting vector whose elements are the element-wise powering operation:

$$r(k) = x(k)^{y(k)}, k = 0{:}n-1$$

- Then, we compute the average for the resulting vector $\vec{r}$.
- For NV sets of two n-element vectors, we get an output vector of NV elements.

Pipelined implementation

- Using *parallel_pipeline*, we define 3 filters and the associated functors (each defined in a class).
  - ✓ First Stage: The parameters passed to the functor are: two 2D data arrays (2 $NV \times n$ matrices), a 2D array, n, and NV. This stage returns two n-element vectors ($\vec{x}$ and $\vec{y}$), and a pointer to store the resulting vector $\vec{r}$ each time.
  - ✓ Second Stage: It computes and returns $r(i) = x(i)^{y(i)}$ for $i = 0, \dots, n-1$. No parameters passed to the functor.
    - We originally attempted to pass the pointer to vector $\vec{r}$ as a parameter (only one array allocated). However, this could potentially create race conditions as two or more instances of the parallel stage would access the same data.
  - ✓ Third Stage: It computes the average of an incoming vector $\vec{r}$ and store the result in an array. The pointer of the resulting array (c) is passed as a parameter to the functor.

- We first show a function where *parallel_pipeline* is invoked:

```
void RunPipeline (int ntoken, int n, int NV, double **a, double **b, double *c, double *p) {
  parallel_pipeline(ntoken,
          make_filter<  void,   MyPair>(filter_mode::serial_in_order, my_in(a,b,r,n,NV) )
       &  make_filter<MyPair,  double*>(filter_mode::parallel,        my_transf() )
       &  make_filter<double*,    void>(filter_mode::serial_in_order, my_out(c,n) ) );
}
```

  - ✓ First Stage (defined by functor `my_in`): Data is stored in 2D arrays `a` and `b`. The stage outputs 3 vectors (`MyPair` class):

```
class my_in {
  double **a;
  double **b;
  double **r;
  int n;
  int NV;
  mutable int i;

public:
  my_in (double **ap, double **bp, double **rp, int np, int NVp): a(ap), b(bp), r(rp),
                                                                  n(np), NV(NVp), i(0) {}
  MyPair operator () (flow_control& fc) const {
    MyPair t;
    const MyPair ret_val = {.x = NULL, .y = NULL, .r=NULL, .n = 0 };
    if (i < NV) {
      t.x = *(a+i); t.y = *(b+i); t.r = *(r+i); t.n = n;
      i++;
      return t; }
    else
      { fc.stop(); return ret_val; }  // 'return ret_val' (NULL values) required in last tbb
  }
};
```

    - `MyPair` class:

```
class MyPair {
public:
  double *x;
  double *y;
  double *r;
  int n;
};
```

    - Note that we use the index `i` (that changes during the execution of the pipeline) to determine if we reached the end of the arrays. The way they are defined, `a` and `b` do not change their values.

  - ✓ Second stage (defined by functor `my_transf`): It performs the element-wise powering operation. As it is a relatively complex operation, it is advantageous to use a parallel stage so that items can be processed concurrently.

```
class my_transf {
public:
  double* operator() (MyPair input) const {
     size_t i;
     double *result = input.r;
     for (i = 0; i < input.n; i++)  result[i] = pow (input.x[i], input.y[i]);
     return result;
  }
};
```

✓ Third Stage (defined by functor `my_out`): It averages the incoming vectors $\vec{r}$ (from the second stage). Syntax-wise, the stage has no outputs, but this stage places the result in the array `c` (provided as an input parameter to its functor).

```
class my_out {
public:
  mutable int j; // so it can be modified by a 'const' operator()
  double *ci;
  int n;
  my_out (double *cp, int np): ci(cp), n(np), j(0) {} // ci = cp, n = np, j=0

  void operator () (double *rt) const {
    size_t k;
    double tmp = 0;

    for (k = 0; k < n; k++) tmp = tmp + rt[k];

    *(ci+j) = tmp/n;
    j++;
  }
};
```

- This is an example of execution of the pipeline:

```
int main () {
  int n = 10; // Length of each vector
  int NV = 20; // # of vectors
  int ntoken = 16;
  double **a, **b, **r, *p;
  int i, j;

  a = (double **) calloc (NV, sizeof(double *));
  b = (double **) calloc (NV, sizeof(double *));
  r = (double **) calloc (NV, sizeof(double *));   // To hold results in Stage 2
  c = (double *) calloc (NV, sizeof(double));

  for (i=0; i < NV; i++) { // 'NV' vectors
     a[i] = (double *) calloc (n, sizeof(double));
     b[i] = (double *) calloc (n, sizeof(double));
     r[i] = (double *) calloc (n, sizeof(double));  }

  for (i=0; i < NV; i++)
    for (j=0; j < n; j++) { a[i][j] = 9.0; b[i][j] = 0.5; }

  RunPipeline (ntoken, n, NV, a, b, r, c);

  cout << "Result:\n"; for (i = 0; i < n; i++) cout << c[i] << "\n";

  free(c);
  for (i = 0; i < n; i++) { free (a[i]); free(b[i]); free(r[i]); }
  free(a); free(b); free(r);
  return 0;
}
```

- Fig. 3 depicts the pipeline and the operations at each stage. Recall that an input item is defined as two n-element vectors.

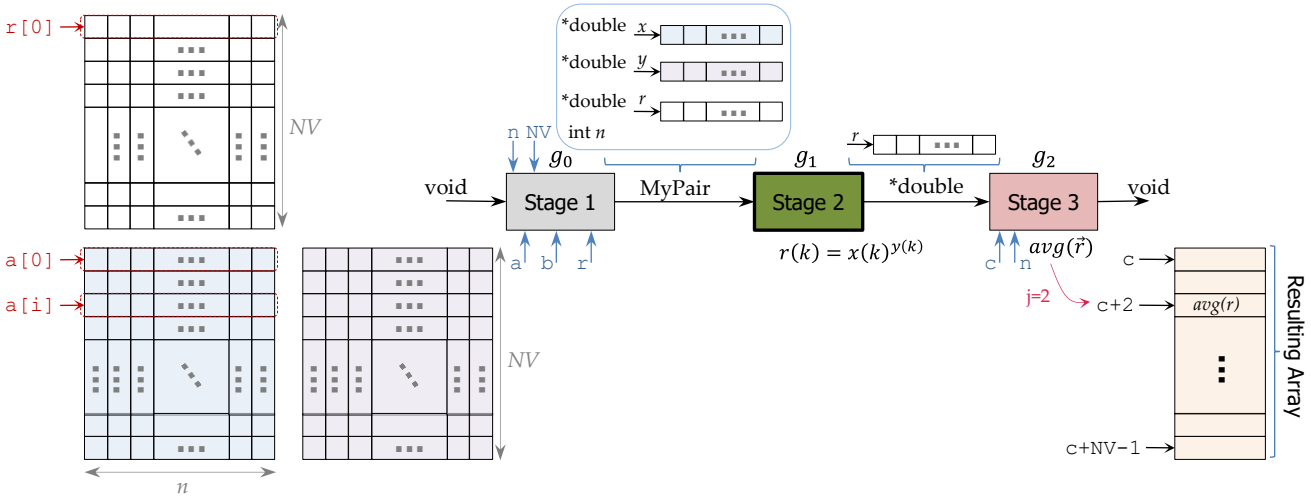

Figure 3. Serial-parallel-serial pipeline. Data is extracted from two 2D arrays. Stage 1 feeds input data (two n-element vectors, and a pointer to an array) into the pipeline. Parallel Stage 2 performs the element-wise powering operation. Stage 3 computes the average of the incoming vector and places the result on an output array.

- The array r ($NV \times n$ elements) is only used so that Stage 1 can pass pointers to a 1-D array. This is needed as we need to store the results of the element-wise powering operation, and the array needs to be allocated.
  - ✓ Note that for every data item, we pass a different pointer to a 1-D array. This is because the 2nd Stage is parallel, and there can be several invocations of it, and each should have their own independent resulting array.

Application files: `pip_avgvec.cpp`

- Compile this application: `g++ pip_avgvec.cpp -ltbb -o pip_avgvec ↵`
- Execute this application: Feel free to modify `n` in order to compute larger sequences
  `./pip_avgvec ↵`