APPENTIX-2
THRUST: a productivity-oriented library for CUDA

```
#include <thrust/host vector.h>
#include <thrust/device vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>

int main(void)
{
     //   generate 16M random numbers on the host

     thrust::host vector<int> h vec(1 << 24);
     thrust::generate(h vec.begin(), h vec.end(), rand);

          //   transfer  data  to  the  device
     thrust::device vector<int>  d_vec  =  h vec;

     //   sort data on the device

     thrust::sort(d vec.begin(), d vec.end());

          //   transfer  data  back  to  host
     thrust::copy(d vec.begin(),  d_vec.end(),  h_vec.begin());

     return  0;
}
```

FIGURE B.1: A complete Thrust program which sorts data on the GPU.

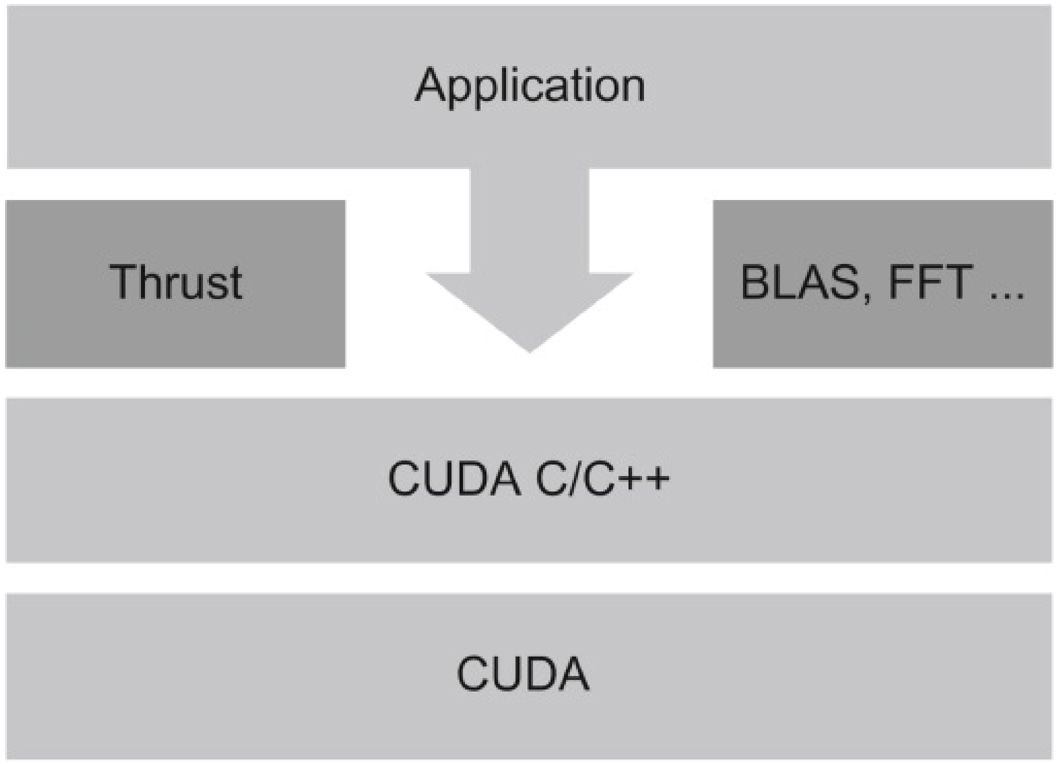


FIGURE B.2: Thrust is an abstraction layer on top of CUDA C/C++.

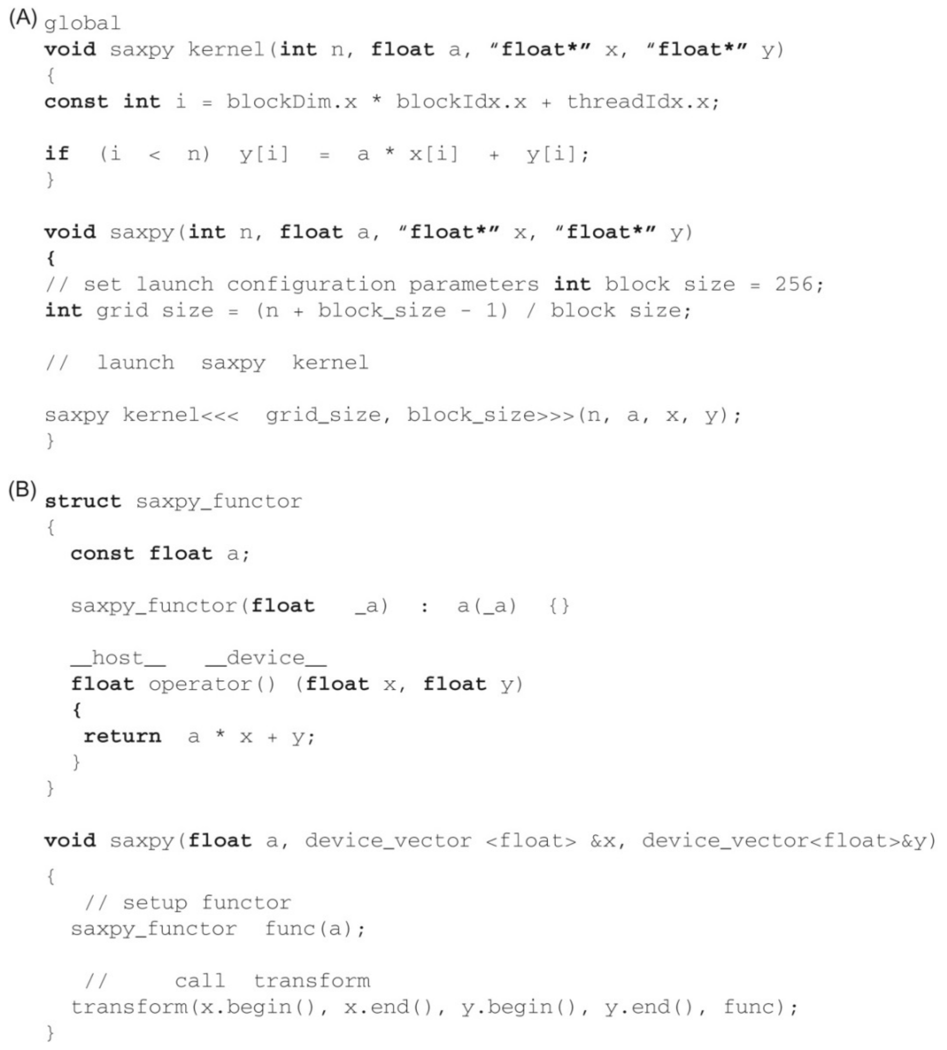
```
size t  N  =  1024;

// allocate Thrust container
device vector< int>  d vec(N);

// extract  raw  pointer  from
container

int raw ptr = raw pointer cast(&d
vec[0]);

// use raw ptr in non Thrustfunctions

cudaMemset(raw ptr,  0,  N
sizeof(int));

// pass raw ptr to a kernel
my kernel<<<N / 128, 128>>>(N, raw
ptr);

// memory is automatically freed
```

(A)

```
size t  N  =  1024;

//  raw  pointer  to  device  memory
int raw ptr;
cudaMalloc(&raw ptr, N  sizeof(int));

// wrap raw pointer with a device ptr
device ptr< int> dev ptr = device
pointer cast(raw ptr);


// use device ptr in Thrust  algorithms
sort(dev ptr, dev ptr + N);


// access device memory through device
ptr
dev ptr[0] = 1;

// free  memory
cudaFree(raw ptr);
```

(B)

FIGURE B.3: Thrust interoperates smoothly with CUDA C/C++. (A) Interfacing Thrust to CUDA and (B) Interfacing CUDA to Thrust.



(A)
```
global
void saxpy kernel(int n, float a, “float*” x, “float*” y)
{
const int i = blockDim.x * blockIdx.x + threadIdx.x;

if  (i  <  n)  y[i]  =  a * x[i]  +  y[i];
}

void saxpy(int n, float a, “float*” x, “float*” y)
{
// set launch configuration parameters int block size = 256;
int grid size = (n + block_size - 1) / block size;

//  launch  saxpy  kernel

saxpy kernel<<<  grid_size, block_size>>>(n, a, x, y);
}
```

(B)
```
struct saxpy_functor
{
  const float a;

  saxpy_functor(float   _a)  :  a(_a)  {}

  __host__   __device__
  float operator() (float x, float y)
  {
   return  a * x + y;
  }
}

void saxpy(float a, device_vector <float> &x, device_vector<float>&y)
{
   // setup functor
  saxpy_functor  func(a);

   //     call  transform
  transform(x.begin(), x.end(), y.begin(), y.end(), func);
}
```

FIGURE B.4: SAXPY implementations in (A) CUDA C and (B) Thrust.

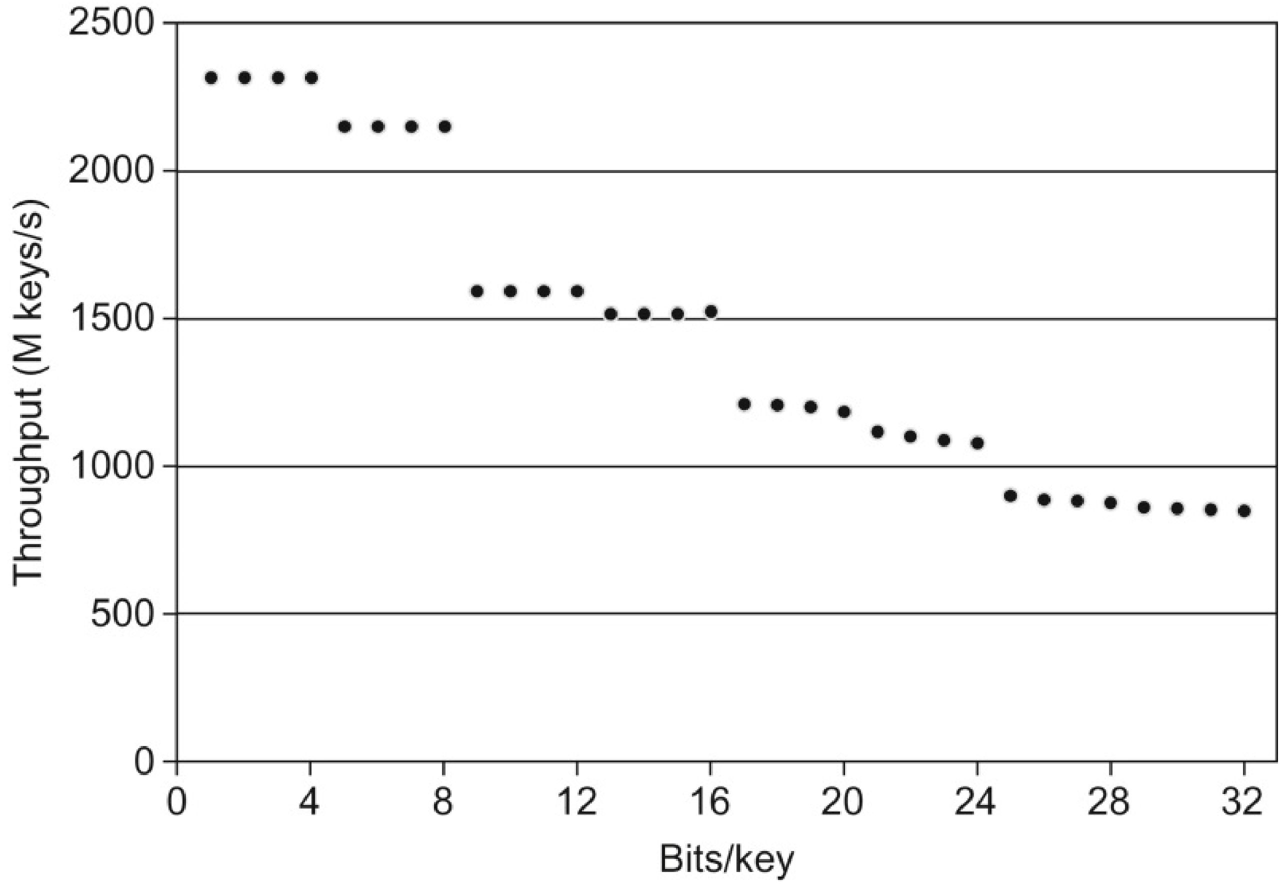


FIGURE B.5: Sorting integers on the GeForce GTX 480: Thrust’s dynamic sorting optimizations improve performance by a considerable margin in common use cases where keys are less than 32 bits.

```
struct square
{

       __host__ __device__
       float operator() (float x) const
       {
       return x *x;
       }
}

float snrm2_slow(const thrust::device vector<float>& x)
{
      //  without  fusion
      device vector<float> temp(x.size()); transform(x.begin(),
      x.end(), temp.begin(), square());

      return  sqrt(  reduce(temp.begin(),  temp.end())  );
}
float snrm2_fast(const thrust::device vector<float>& x)
{
      //  with  fusion
      return  sqrt(  transform_reduce(x.begin(),x.end(),square(),0.0f,
      plus<float>());
}
```

FIGURE B.6: SNRM2 has low arithmetic intensity and therefore benefits greatly from fusion.

```
struct  float3
{

   float  x;
   float  y;
   float  z;
}
float3  aos[100];
      ...
aos[0].x  =  1.0f;
```

(A)

```
struct  float3_soa

{

   float     x[100];
   float     y[100];
   float     z[100];
}
float3_soa soa;
      ...
soa.x[0]  =  1.0f;
```

(B)

FIGURE B.7: Data layouts for three-dimensional float vectors. (A) Array of structures and (B) structure of arrays.

```
struct rotate tuple {
   __host__ __device__
   tuple<float,float,float> operator()(tuple<float,float,float>& t) {
      float x = get<0>(t);
      float y = get<1>(t);
      float z = get<2>(t);
      float rx = 0.36f * x + 0.48f * y + 0.80f * z;
      float ry = 0.80f * x + 0.60f * y + 0.00f * z;
      float rz = 0.48f * x + 0.64f * y + 0.60f * z;
       return make_tuple(rx, ry, rz);
  }
};
device vector<float> x(N), y(N), z(N);
transform(make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
     make_zip_iterator(make_tuple(x.end(), y.end(), z.end())),
     make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
     rotate tuple());
```

FIGURE B.8: The zip iterator facilitates processing of data in structure of arrays format.

```
struct smaller_tuple {
   tuple<float,int> operator()(tuple<float,int> a,tuple<float,int> b) {
       // return the tuple with the smaller float value
       if (get<0>(a) < get<0>(b)) return a;
       else  return b;
    }
};

int min_index(device vector<float>& values) {
  // [begin,end) form the implicit sequence [0,1,2, ... value.size())
   counting iterator<int> begin(0);
   counting iterator<int> end(values.size());

  // initial value of the reduction
   tuple<float,int> init(values[0], 0);

  // compute the smallest tuple
  tuple<float,int> smallest =
     reduce(make_zip_iterator(make_tuple(values.begin(), begin)),
     make_zip_iterator(make_tuple(values.end(), end)),
     init, smaller_tuple());
   // return the index
   return get<1>(smallest);
}
```

FIGURE B.9: Implicit ranges improve performance by conserving memory bandwidth.