APPENTIX-4
An introduction to C++ AMP

```
__global__void vecAddKernel(float* d_A, float* d_B, float* d_C, int n)
{
        int i = blockDim.x * blockIdx.x + threadIdx.x;
       if (i < n) C[i] = A[i] + B[i];
}
void vecAdd(float* A, float* B, float* C, int n)
{
        int size = n * sizeof (float); float* d_A, d_B, d_C;
        cudaMalloc((void **) &d_A, size);
        cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &d_B, size);
        cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
        cudaMalloc((void **) &d_C, size);
        vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
        cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
        cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

FIGURE D.1: CUDA vector addition from Chapter 2, Data parallel computing.

```
1  #include <amp.h>
2  using namespace concurrency;
3
4  void vecAdd(float* A, float* B, float* C, int n)
5  {
6      array_view<const float,1> AV(n,A), BV(n,B);
7      array_view<float,1> CV(n,C);
8      CV.discard_data();
9      parallel_for_each(CV.extent, [=](index<1> i) restrict(amp)
10     {
11         CV[i] = AV[i] + BV[i];
12     });
13     CV.synchronize();
14 }
```

FIGURE D.2: Vector addition in C++ AMP.

```
#include <amp_math.h>
void cenergy_2(float * energygrid, extent<3> grid,
               float gridspacing, float z, int k,
               const float * atoms, int numatoms) {
    array_view<float,3> energygrid_view(grid, energygrid);
    array_view<float,2> energy_slice = energygrid_view(k);
    energy_slice.discard_data();
    array_view<const float,2> atom_view(numatoms,4,atoms);
    parallel_for_each(energy_slice.extent, [=](index<2> ji)
    restrict(amp) {
          float y = gridspacing * float(ji[0]);
          float x = gridspacing * float(ji[1]);
          float energy = 0.0f;
          for(int n =0; n < numatoms; n++) {
               float dx = x - atom_view(n,0);
               float dy = y - atom_view(n,1);
               float dz = z - atom_view(n,2);
               energy + = atom_view(n,3)/
                          precise_math::sqrtf(dx*dx + dy*dy+dz*dz);
          }
          energy_slice[ji] = energy;
    });
    energy_slice.synchronize();
}
```

FIGURE D.3: Base Coulomb potential calculation.

```
1  void vecAdd(float* A, float* B, float* C, int n)
2  {
3      array<float,1> AA(n), BA(n);
4      array<float,1> CA(n);
5      copy(A,AA);
6      copy(B,BA);
7      parallel_for_each(CA.extent,
8              [&AA,&BA,&CA](index<1> i) restrict(amp)
9      {
10        CA[i] = AA[i] + BA[i];
11     });
12     copy(CA,C);
13 }
```

FIGURE D.4: Explicit memory and copy management.

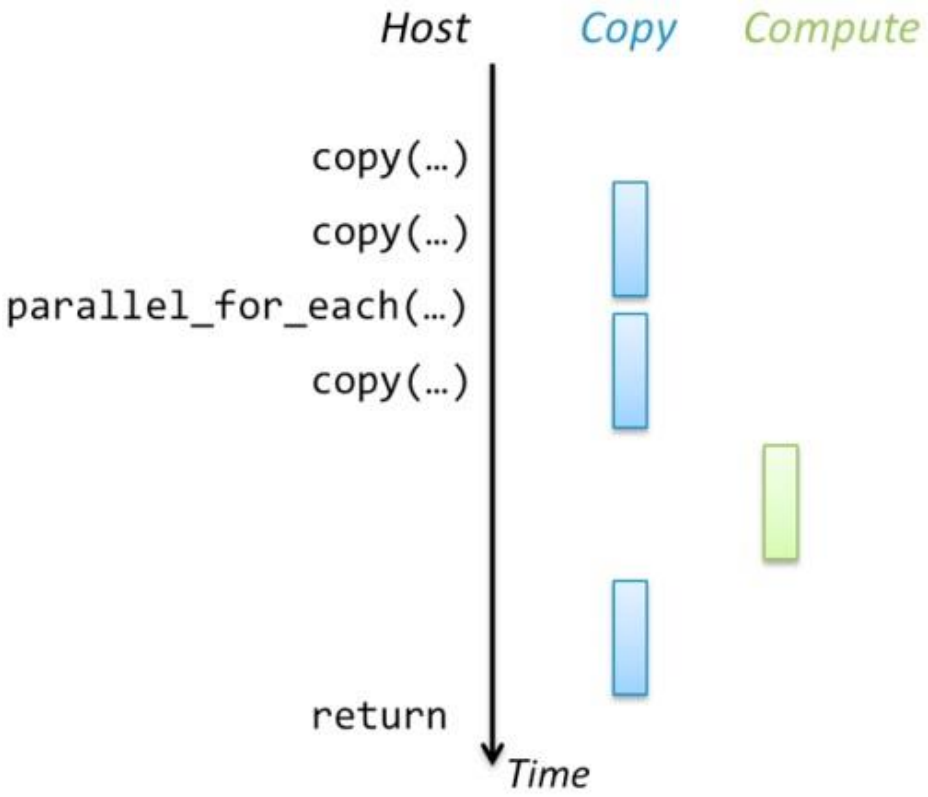


FIGURE D.5: Concurrent host/accelerator execution.

```
1 parallel_for_each(CV.extent, [=](index<1> i) restrict(amp)
2 {
3         CV[i] = AV[i] + BV[i];
4 });
5 completion_future done = CV.synchronize_async();
6 otherProcessing(A,B);
7 done.get();
```

FIGURE D.6: Overlapped accelerator and host processing.

```
1  accelerator find_accelerator() {
2      vector<accelerator> accs = accelerator::get_all();
3      auto result =
4          find_if(accs.begin(), accs.end(), [](const accelerator& acc)
5          {
6              return acc.supports_double_precision &&
7                     !acc.has_display;
8          });
9      if(result == accs.end())
10         throw std::string("No suitable accelerator found");
11     return *result;
12 }
```

FIGURE D.7: Example of finding an accelerator.

```
1   void vecAdd (float* A, float* B, float* C, intn)
2   {
3           accelerator acc;
4           accelerator_view view(acc.default_view);
5           array<float,1> AA(n,view), BA(n,view);
6           array<float,1> CA(n,view);
7           copy(A,AA);
8           copy(B,BA);
9           parallel_for_each(view, CA.extent,
10                [&AA,&BA,&CA](index<1> i) restrict(amp)
11          {
12             CA[i] = AA[i] + BA[i];
13          });
14          copy(CA,C);
15  }
```

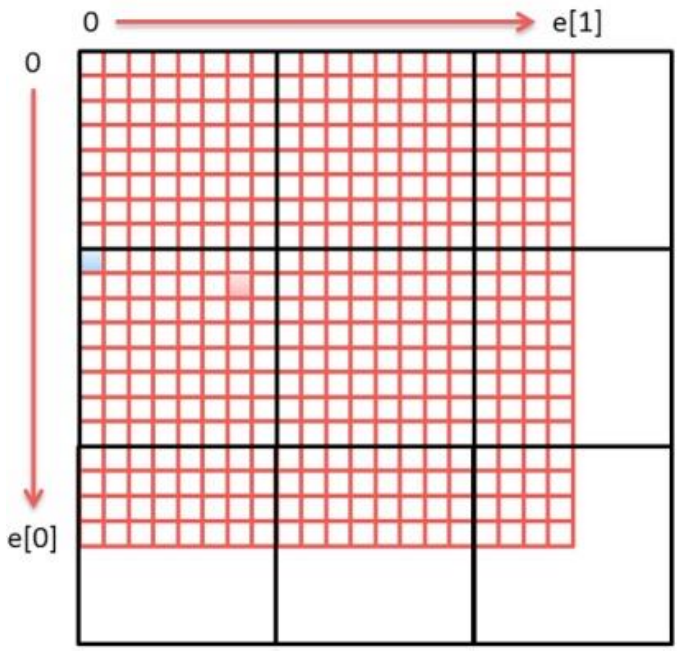
FIGURE D.8: Explicit accelerator use.

```
1  using std::vector;
2  void vecAddLong(float *A, float *B, float *C, int n,
3                  accelerator_view acc)
4  {
5      int block = (acc.accelerator.dedicated_memory * 1024)
6                              /(3*sizeof(float));
7      vector<completion_future> results;
8      for(int i = 0; i < n; i += block) {
9          int m = min(n-i,block);
10         array_view<const float,1> AV(m,A+i), BV(m,B+i);
11         array_view<float,1> CV(m,C+i);
12         CV.discard_data();
13         parallel_for_each(acc, CV.extent, [=](index<1> idx) restrict(amp)
14         {
15             CV[idx] = AV[idx] + BV[idx];
16         });
17         results.push_back(CV.synchronize_async());
18     }
19     std::for_each(results.begin(), results.end(),
20                   [](completion_future f) { f.get(); });
21 }
```

FIGURE D.9: Explicit accelerator with asynchronous transfers.

```
1  void MatrixMul(float * M, float * N, float *P, int Width) {
2      extent<2> dims(Width,Width);
3      array_view<const float,2> d_M(dims,M), d_N(dims,N);
4      array_view<float,2> d_P(dims,P);
5      d_P.discard_data();
6      tiled_extent<TILE_WIDTH,TILE_WIDTH> tiled(dims);
7      parallel_for_each(tiled,
8           [=](tiled_index<TILE_WIDTH,TILE_WIDTH> t_idx) restrict(amp){
9             tile_static float Mds[TILE_WIDTH][TILE_WIDTH];
10            tile_static float Nds[TILE_WIDTH][TILE_WIDTH];
11            int tx  = t_idx.local[0],  ty  = t_idx.local[1];
12            int Row = t_idx.global[0], Col = t_idx.global[1];
13            float Pvalue = 0;
14            for (int m = 0; m < Width/TILE_WIDTH; ++m) {
15                Mds[tx][ty] = d_M(m*TILE_WIDTH+tx, Row);
16                Nds[tx][ty] = d_N(Col, m*TILE_WIDTH+ty);
17                t_idx.barrier.wait();
18                for(int k = 0; k < TILE_WIDTH; k++)
19                       Pvalue += Mds[tx][k] * Mds[k][ty];
20                t_idx.barrier.wait();
21            }
22            d_P(Row,Col) = Pvalue;
23     });
24     d_P.synchronize();
25 }
```

FIGURE D.10: Tiled matrix multiplication.




```
extent<2> e(20,20);
tiled_extent<8,8> te(e);
auto te2 = e.tile<8,8>();
auto pte = te.pad();
auto tte = te.truncate();
parallel_for_each(pte,
    [](tiled_index<8,8> t_idx)
    restrict(amp)
{
    t_idx.global; // 9,6 for example
    t_idx.tile_extent ; // 3,3
    t_idx.tile    ; // 1,0
    t_idx.tile_origin ; // 8,0
    t_idx.local   ; // 1,6
 });
```

FIGURE D.11: Illustration of tiling 20× 20 compute domain.

```
1  #include <amp_graphics.h>
2  using namespace graphics;
3  …
4          norm n;          // normalized types
5          unorm u;
6          float_2 f2;
7          float_4 f4;      // short vector types;
8          int_2 i2;
9          norm_2 n2;
10         f2 = f4.xy + i2.x*f4.zw;
11         // usable in arrays, array_views
12         extent<2> e(1024,1024);
13         array<norm_2,2> an2(e);
14         // and in textures
15         texture<unorm_4,2> tu2(e, data, e.size() * 16U, 16U);
16         writeonly_texture_view<unorm_4,2> wotv(tu2);
```

FIGURE D.12: Examples of type from concurrency::graphics.

```
1  struct Vertex2D { float_2 Pos; };
2  IUnknown * my_rotate(ID3D11Device* d3ddevice, float THETA,
3                       int num_elements, const float_2 * data)
4  {
5      // copy data into a DX buffer
6      accelerator_view acc = create_accelerator_view(d3ddevice);
7      array<Vertex2D,1> vertices(num_elements, data, acc);
8      parallel_for_each(vertices.extent,
9                        [=, &vertices] (index<1> idx) restrict(amp) {
10         // Rotate the vertex by angle THETA
11         float_2 pos = vertices[idx].Pos;
12         vertices[idx].Pos.y = pos.y * cos(THETA) - pos.x * sin(THETA);
13         vertices[idx].Pos.x = pos.y * sin(THETA) + pos.x * cos(THETA);
14     });
15     // return the DX buffer use of transformed data.
16     return get_buffer(vertices);
17 }
```

FIGURE D.13: Examples DirectX interop—rotate vertex list.