Turning FPGAs Into Supercomputers

- Debunking the Myths About FPGA-based Software Acceleration

Anders Dellson*
Göran Sandberg
Stefan Möhl

mitrion

Conclusion

- The promises of FPGA Supercomuting are real for many applications
- What used to be obstacles using older technology are now - myths
- It's easy to evaluate FPGA feasibility, and to build and maintain HPC production environments using the Mitrion platform

The Promises of FPGA Supercomputing

Compared to CPUs:

- Order/s of magnitude performance gain per chip
- Very low power consumption per GOPs
- FPGAs will continue to ride Moore’s Law

\- You heard all this before.

mitrion

The Obstacles

1. Electrical Engineering skills are necessary to program FPGAs
2. Application development is complex and time-consuming
3. A big initial investment is required in FPGA computers and EDA tool seats
4. Lack of portability across FPGA generations and FPGA computers

mitrion

~~The Obstacles~~ Myths

1. Electrical Engineering skills are necessary to program FPGAs
2. Application development is complex and time-consuming
3. A big initial investment is required in FPGA computers and EDA tool seats
4. Lack of portability across FPGA generations and FPGA computers

mitrion

Part I

Clarifying a few points of confusion

Field Programmable Gate Arrays *Are Not Programmable (!)*

- Without a circuit design, an FPGA is just an empty silicon surface
- What is meant by “Programmable” in the acronym “FPGA” is “a circuit design can be loaded”
- Designing a circuit is *not* “programming” from a software developer’s point of view

mitrion

Hardware versus Software - a Culture Clash

- Hardware design
  - Driven by the design cycle
  - Silicon cost – size and speed
  - Precise control of electrical signals
- Software design
  - Driven by the code-base life cycle
  - Development cost – code maintenance
  - Abstract description of algorithm

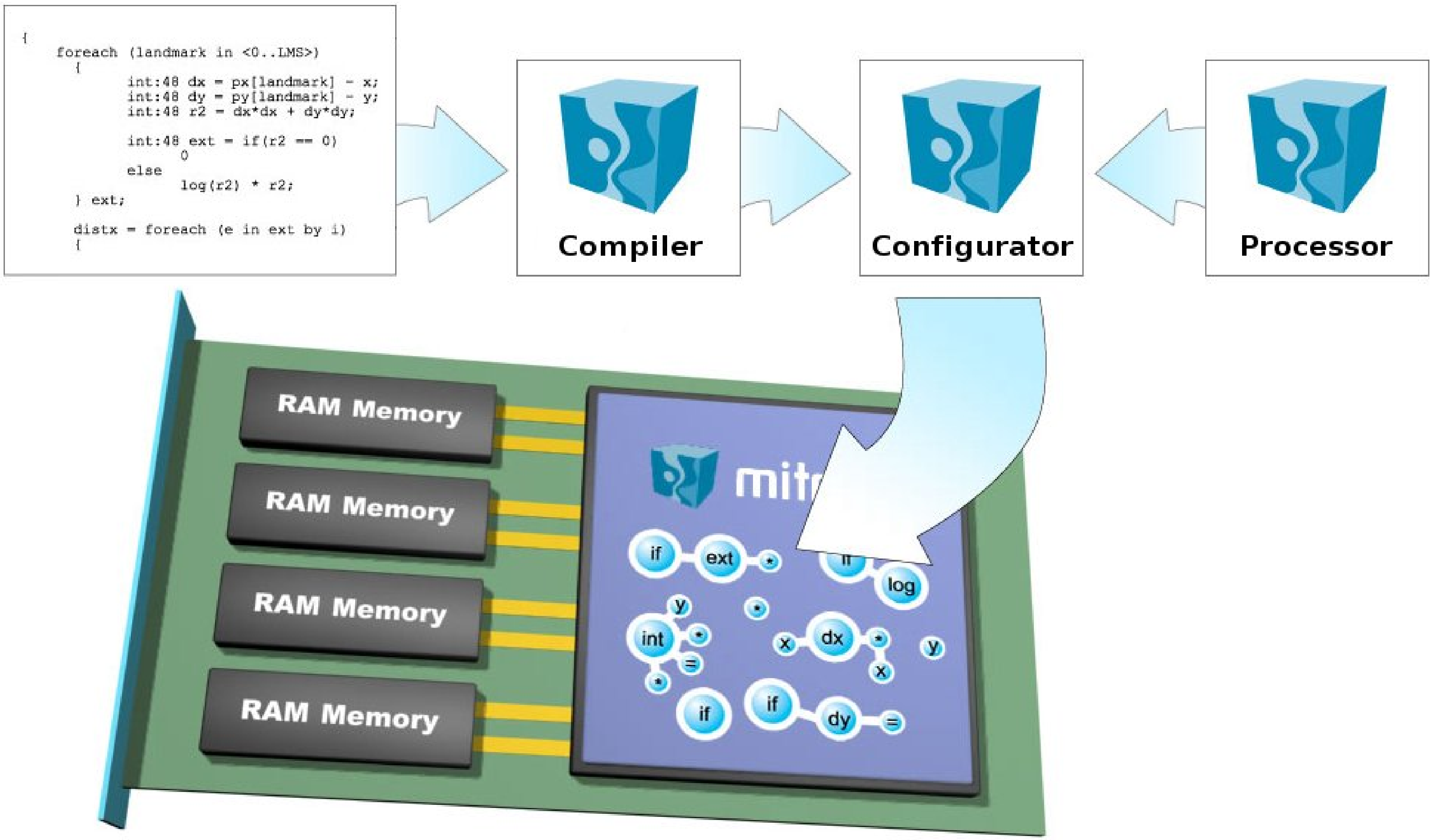
mitrion

An FPGA in a Supercomputer Fundamentally Changes the Conditions

- Supercomputing users are not EEs
  - Biologists, Astronomers, Chemists, etc...
- The FPGA is unspecific
  - Instead of one design – many chips – we have one chip – many designs
- The FPGA is reconfigurable
  - Design for program life cycle, not design cycle
- The FPGA is fixed
  - Use as much space and speed as the FPGA allows

mitrion

The Mitrion Platform



The Essential Parts of the Mitrion Platform

- The Mitrion Virtual Processor
  - A fine-grain massively parallel, configurable soft-core processor for FPGAs
- The Mitrion-C programming language
  - An intrinsically parallel C-family language
- The Mitrion Software Development Kit
  - Compiler
  - Debugger/Simulator
  - Processor configurator

mitrion

Part II

Debunking the Myths

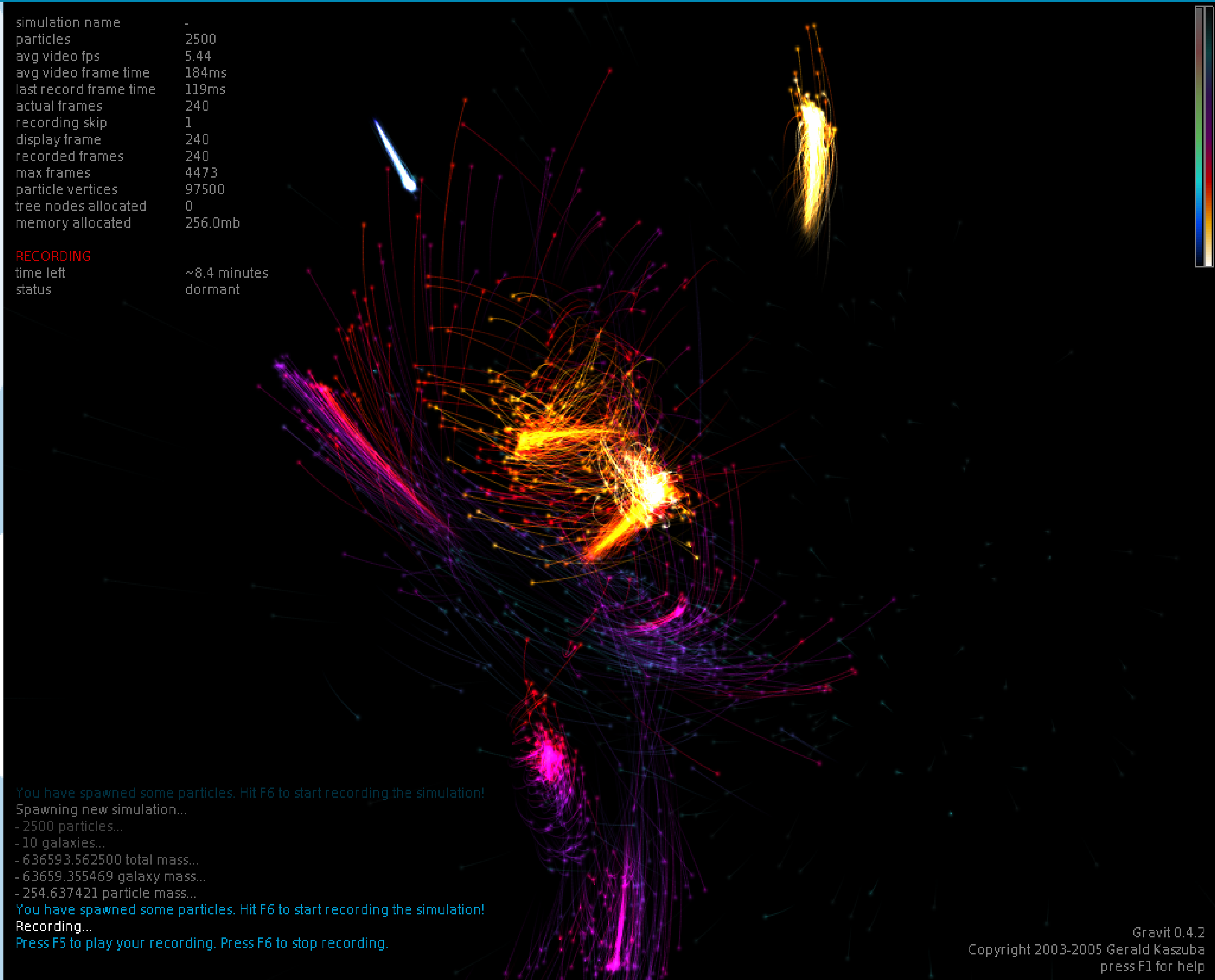
~~The Obstacles~~ Myths

1. Electrical Engineering skills are necessary to program FPGAs
2. Application development is complex and time-consuming
3. A big initial investment is required in FPGA computers and EDA tool seats
4. Lack of portability across FPGA generations and FPGA computers

mitrion

The Application: *Gravit*

- N-body gravity simulator
- Open source



Step 1: Identify FPGA Part

```
nbody(float *x,   // input vectors
      float *y,
      float *z,
      float *mass,

      float *fx,  // output vectors
      float *fy,
      float *fz)
{

    int i;
    int j;

    bzero(fx, sizeof(float)*PARTICLES);
    bzero(fy, sizeof(float)*PARTICLES);
    bzero(fz, sizeof(float)*PARTICLES);

    for( j = 0; j<PARTICLES; j++)
    {
        for( i = 0; i<PARTICLES; i++)
        {
            if(i != j)
            {
                float dx = x[i] - x[j];
                float dy = y[i] - y[j];
                float dz = z[i] - z[j];

                float d = dx*dx + dy*dy + dz*dz;
                float force = (-0.000010f * mass[i] * mass[j]) / d;

                fx[j] += force * dx;
                fy[j] += force * dy;
                fz[j] += force * dz;
            }
        }
    }
}
```

- Download source code from the Internet
- Indentify compute-intensive kernel to run on FPGA in Mitrion processor

Step 1: Identify FPGA Part

```
nbody(float *x,   // input vectors
      float *y,
      float *z,
      float *mass,

      float *fx,  // output vectors
      float *fy,
      float *fz)
{

    int i;
    int j;

    bzero(fx, sizeof(float)*PARTICLES);
    bzero(fy, sizeof(float)*PARTICLES);
    bzero(fz, sizeof(float)*PARTICLES);

    for( j = 0; j<PARTICLES; j++)
    {
        for( i = 0; i<PARTICLES; i++)
        {
            if(i != j)
            {
                float dx = x[i] - x[j];
                float dy = y[i] - y[j];
                float dz = z[i] - z[j];

                float d = dx*dx + dy*dy + dz*dz;
                float force = (-0.000010f * mass[i] * mass[j]) / d;

                fx[j] += force * dx;
                fy[j] += force * dy;
                fz[j] += force * dz;
            }
        }
    }
}
```

- Download source code from the Internet
- Indentify compute intensive kernel to run on FPGA in Mitrion processor

The computationally intense part is this double loop in the nbody function

Step 2: Replace With Function Call to FPGA

```
nbody(float *x,   // input vectors
      float *y,
      float *z,
      float *mass,

      float *fx,  // output vectors
      float *fy,
      float *fz)
{
    int i;

    // Store positions and masses in FPGA RAM banks
    for( i = 0; i<PARTICLES; i++)
    {
        int off = i*4;
        ram[off+0] = x[i];
        ram[off+1] = y[i];
        ram[off+2] = z[i];
        ram[off+3] = mass[i];
    }

    // Start the Mitrion Virtual Processor
    mitrion_processor_run(p);
    // The run function is asynchronous, so we have to wait
    // explicitly. This call blocks until the MVP has finished.
    mitrion_processor_wait(p);


    // Read results back from FPGA RAMs
    for( i = 0; i<PARTICLES; i++)
    {
        int off = i*4;
        fx[i] = result_ram[off+0];
        fy[i] = result_ram[off+1];
        fz[i] = result_ram[off+2];
    }

}
```

- API calls are available to initialize and control the FPGA.

Step 2: Replace With Function

```
// Store positions and masses in FPGA RAM banks
for( i = 0; i<PARTICLES; i++)
{
    int off = i*4;
    ram[off+0] = x[i];
    ram[off+1] = y[i];
    ram[off+2] = z[i];
    ram[off+3] = mass[i];
}
```


```
// Store positions and masses in FPGA RAM banks
for( i = 0; i<PARTICLES; i++)
{
    int off = i*4;
    ram[off+0] = x[i];
    ram[off+1] = y[i];
    ram[off+2] = z[i];
    ram[off+3] = mass[i];
}
```


Manage data transfers to local FPGA memories

```
// Read results back from FPGA RAMs
for( i = 0; i<PARTICLES; i++)
{
    int off = i*4;
    fx[i] = result_ram[off+0];
    fy[i] = result_ram[off+1];
    fz[i] = result_ram[off+2];
}
```


```
// Read results back from
for( i = 0; i<PARTICLES; i
{
    int off = i*4;
    fx[i] = result_ram[off
    fy[i] = result_ram[off
    fz[i] = result_ram[off
}

}
```

Step 2: Replace With Function Call to FPGA

- API calls are available to initialize and control the FPGA.

```
// Start the Mitrion Virtual Processor
mitrion_processor_run(p);
// The run function is asynchronous, so we have to wait
// explicitly. This call blocks until the MVP has finished.
mitrion_processor_wait(p);
```

Replace loop with function call to FPGA

// Start the Mitrion
mitrion_processor_ru
// The run function
// explicitly. This
mitrion_processor_wa

Myth #1:

Electrical Engineering skills are necessary to program FPGAs

Step 3: Rewrite Kernel in Mitrion-C

```
nbody(float *x,   // input vectors
      float *y,
      float *z,
      float *mass,

      float *fx,  // output vectors
      float *fy,
      float *fz)
{

  int i;
  int j;

  bzero(fx, sizeof(float)*PARTICLES);
  bzero(fy, sizeof(float)*PARTICLES);
  bzero(fz, sizeof(float)*PARTICLES);

  for( j = 0; j<PARTICLES; j++)
  {



    for( i = 0; i<PARTICLES; i++)
    {

      if(i != j)
      {
        float dx = x[i] - x[j];
        float dy = y[i] - y[j];
        float dz = z[i] - z[j];

        float d = dx*dx + dy*dy + dz*dz;
        float force = (-0.000010f * mass[i] * mass[j]) / d;

        fx[j] += force * dx;
        fy[j] += force * dy;
        fz[j] += force * dz;
      }


    }
  }






}
```

- Use original algorithm as a starting point

Step 3: Rewrite Kernel in Mitrion-C

```
(ExtRAM, ExtRAM, ExtRAM, ExtRAM)
    main (ExtRAM ram0, ExtRAM ram1, ExtRAM ram2, ExtRAM ram3)
{





  Float<PARTICLES> final_fx;
  Float<PARTICLES> final_fy;
  Float<PARTICLES> final_fz;

  Float<PARTICLES> fx = foreach(e in <1.. PARTICLES>) 0.0;
  Float<PARTICLES> fy = foreach(e in <1.. PARTICLES>) 0.0;
  Float<PARTICLES> fz = foreach(e in <1.. PARTICLES>) 0.0;

  (final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES>)
  {
     i = e-1;
    (x_i,y_i,z_i,mass_i) = read_particle(ram0, i);

    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in  fx,   fy,   fz   by j)
    {
      (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
      (new_fx_j, new_fy_j, new_fz_j) = if(i != j)
      {
        Float dx = x_j - x_i;
        Float dy = y_j - y_i;
        Float dz = z_j - z_i;

        Float d = dx*dx + dy*dy + dz*dz;
        Float force = -0.000010 * mass_i * mass_j / d;

        Float x = fx_j + force * dx;
        Float y = fy_j + force * dy;
        Float z = fz_j + force * dz;
      } (x,y,z)
      else
      { } (fx_j, fy_j, fz_j);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);

  ram1_3 = foreach(fx, fy, fz in final_fx, final_fy, final_fz by i)
  {
     ram1_2 = write_particle_force(ram1, i, fx, fy, fz);
  } ram1_2;

} (ram0, ram1_3, ram2, ram3);
```

- Use original algorithm as a starting point
- Mitrion-C version very similar, but not yet optimized for speed

mitrion

```
(final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES>)
  {
     i = e-1;
    (x_i,y_i,z_i,mass_i) = read_particle(ram0, i);

    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in  fx,   fy,   fz   by j)
    {
      (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
      (new_fx_j, new_fy_j, new_fz_j) = if(i != j)
      {
        Float dx = x_j - x_i;
        Float dy = y_j - y_i;
        Float dz = z_j - z_i;

        Float d = dx*dx + dy*dy + dz*dz;
        Float force = -0.000010 * mass_i * mass_j / d;

        Float x = fx_j + force * dx;
        Float y = fy_j + force * dy;
        Float z = fz_j + force * dz;
      } (x,y,z)
      else
      { } (fx_j, fy_j, fz_j);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);
```

Mitrion-C code structure identical to original C code


```
(final_fx, final
{
   i = e-1;
  (x_i,y_i,z_i,m

  (fx, fy, fz)=
  {
    (x_j, y_j, z
    (new_fx_j, n
    {
      Float dx =
      Float dy =
      Float dz =

      Float d =
      Float forc

      Float x = fx_j + force * dx;
      Float y = fy_j + force * dy;
      Float z = fz_j + force * dz;
    } (x,y,z)
    else
    { } (fx_j, fy_j, fz_j);
  } (new_fx_j, new_fy_j, new_fz_j);
} (fx, fy, fz);
```


mitrion

```
(final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES>)
  {
     i = e-1;
    (x_i,y_i,z_i,mass_i) = read_particle(ram0, i);

    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in  fx,   fy,   fz   by j)
    {
      (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
      (new_fx_j, new_fy_j, new_fz_j) = if(i != j)
      {
        Float dx = x_j - x_i;
        Float dy = y_j - y_i;
        Float dz = z_j - z_i;

        Float d = dx*dx + dy*dy + dz*dz;
        Float force = -0.000010 * mass_i * mass_j / d;

        Float x = fx_j + force * dx;
        Float y = fy_j + force * dy;
        Float z = fz_j + force * dz;
      } (x,y,z)
      else
      { } (fx_j, fy_j, fz_j);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);
```

Major difference - data is read from RAM memory

Mitrion-C code structure identical to original C code


```
(final_fx, final
{
   i = e-1;
  (x_i,y_i,z_i,m

  (fx, fy, fz)=
  {
    (x_j, y_j, z
    (new_fx_j, n
    {
      Float dx =
      Float dy =
      Float dz =

      Float d =
      Float forc

      Float x = fx_j + force * dx;
      Float y = fy_j + force * dy;
      Float z = fz_j + force * dz;
    } (x,y,z)
    else
    { } (fx_j, fy_j, fz_j);
  } (new_fx_j, new_fy_j, new_fz_j);
} (fx, fy, fz);
```

Step 3: Rewrite Kernel in Mitrion-C

```
(ExtRAM, ExtRAM, ExtRAM, ExtRAM)
    main (ExtRAM ram0, ExtRAM ram1, ExtRAM ram2, ExtRAM ram3)
{





  Float<PARTICLES> final_fx;
  Float<PARTICLES> final_fy;
  Float<PARTICLES> final_fz;

  Float<PARTICLES> fx = foreach(e in <1.. PARTICLES>) 0.0;
  Float<PARTICLES> fy = foreach(e in <1.. PARTICLES>) 0.0;
  Float<PARTICLES> fz = foreach(e in <1.. PARTICLES>) 0.0;

  (final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES>)
  {
     i = e-1;
    (x_i,y_i,z_i,mass_i) = read_particle(ram0, i);

    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in  fx,   fy,   fz   by j)
    {
      (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
      (new_fx_j, new_fy_j, new_fz_j) = if(i != j)
      {
        Float dx = x_j - x_i;
        Float dy = y_j - y_i;
        Float dz = z_j - z_i;

        Float d = dx*dx + dy*dy + dz*dz;
        Float force = -0.000010 * mass_i * mass_j / d;

        Float x = fx_j + force * dx;
        Float y = fy_j + force * dy;
        Float z = fz_j + force * dz;
      } (x,y,z)
      else
      { } (fx_j, fy_j, fz_j);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);

  ram1_3 = foreach(fx, fy, fz in final_fx, final_fy, final_fz by i)
  {
     ram1_2 = write_particle_force(ram1, i, fx, fy, fz);
  } ram1_2;

} (ram0, ram1_3, ram2, ram3);
```

- Use original algorithm as a starting point
- Mitrion-C version very similar, but not yet optimized for speed
- No hardware design considerations

mitrion

Step 3: Rewrite Kernel in Mitrion-C

```
(ExtRAM, ExtRAM, ExtRAM, ExtRAM)
    main (ExtRAM ram0, ExtRAM ram1, ExtRAM ram2, ExtRAM ram3)
{






  Float<PARTICLES> final_fx;
  Float<PARTICLES> final_fy;
  Float<PARTICLES> final_fz;

  Float<PARTICLES> fx = foreach(e in <1.. PARTICLES>) 0.0;
  Float<PARTICLES> fy = foreach(e in <1.. PARTICLES>) 0.0;
  Float<PARTICLES> fz = foreach(e in <1.. PARTICLES>) 0.0;

  (final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES>)
  {
     i = e-1;
    (x_i,y_i,z_i,mass_i) = read_particle(ram0, i);

    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in  fx,   fy,   fz   by j)
    {
      (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
      (new_fx_j, new_fy_j, new_fz_j) = if(i != j)
      {
        Float dx = x_j - x_i;
        Float dy = y_j - y_i;
        Float dz = z_j - z_i;

        Float d = dx*dx + dy*dy + dz*dz;
        Float force = -0.000010 * mass_i * mass_j / d;

        Float x = fx_j + force * dx;
        Float y = fy_j + force * dy;
        Float z = fz_j + force * dz;
      } (x,y,z)
      else
      { } (fx_j, fy_j, fz_j);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);

  ram1_3 = foreach(fx, fy, fz in final_fx, final_fy, final_fz by i)
  {
     ram1_2 = write_particle_force(ram1, i, fx, fy, fz);
  } ram1_2;

} (ram0, ram1_3, ram2, ram3);
```

- Use original algorithm as a starting point
- Mitrion-C version very similar, but not yet optimized for speed
- No hardware design considerations
- ~~**Myth #1: Electrical Engineering skills are necessary to program FPGAs**~~

mitrion

Myth #2:

Application development is complex and time-consuming

mitrion

Step 4: Optimize the Mitrion-C Code for Increased Performance

```
...
  (final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES_DIV>)
  {
     i = (e-1)*4;
     // read 4 particles into lists
     (x_il,y_il,z_il,mass_il) = foreach(v in <0..3>)
     {
         (x,y,z,mass) = read_particle(ram0, i+v);
     } (x,y,z,mass);
     x_iv    = reformat(x_il, [4]);
     y_iv    = reformat(y_il, [4]);
     z_iv    = reformat(z_il, [4]);
     mass_iv = reformat(mass_il, [4]);
    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in
                          fx,   fy,   fz   by j)
    {
       (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
        // match with 4 particles at a time
        (fx_ijv, fy_ijv, fz_ijv) = foreach (x_i, y_i, z_i,  mass_i in
                                            x_iv,y_iv,z_iv, mass_iv by v)
        {
            (fx_ij, fy_ij, fz_ij) = if(i != j)
            {
                Float dx = x_j - x_i;
                Float dy = y_j - y_i;
                Float dz = z_j - z_i;

                Float d = dx*dx + dy*dy + dz*dz;
                Float force = -0.000010 * mass_i * mass_j / d;

                Float x = force * dx;
                Float y = force * dy;
                Float z = force * dz;
            } (x,y,z)
            else
            { zero = 0;} (zero,zero,zero);
        } (fx_ij, fy_ij, fz_ij);
        new_fx_j = fx_j + sum4v(fx_ijv);
        new_fy_j = fy_j + sum4v(fy_ijv);
        new_fz_j = fz_j + sum4v(fz_ijv);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);
...
```

- Use your parallel programming skills to optimize performance

Step 4: Optimize the Mitrion-C Code for Increased Performance

```
...
  (final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES_DIV>)
  {
     i = (e-1)*4;
     // read 4 particles into lists
     (x_il,y_il,z_il,mass_il) = foreach(v in <0..3>)
     {
         (x,y,z,mass) = read_particle(ram0, i+v);
     } (x,y,z,mass);
     x_iv    = reformat(x_il, [4]);
     y_iv    = reformat(y_il, [4]);
     z_iv    = reformat(z_il, [4]);
     mass_iv = reformat(mass_il, [4]);
    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in
                          fx,   fy,   fz   by j)
    {
      (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
       // match with 4 particles at a time
       (fx_ijv, fy_ijv, fz_ijv) = foreach (x_i, y_i, z_i, mass_i in
                                           x_iv,y_iv,z_iv, mass_iv by v)
       {
           (fx_ij, fy_ij, fz_ij) = if(i != j)
           {
               Float dx = x_j - x_i;
               Float dy = y_j - y_i;
               Float dz = z_j - z_i;

               Float d = dx*dx + dy*dy + dz*dz;
               Float force = -0.000010 * mass_i * mass_j / d;

               Float x = force * dx;
               Float y = force * dy;
               Float z = force * dz;
           } (x,y,z)
           else
           { zero = 0;} (zero,zero,zero);
       } (fx_ij, fy_ij, fz_ij);
       new_fx_j = fx_j + sum4v(fx_ijv);
       new_fy_j = fy_j + sum4v(fy_ijv);
       new_fz_j = fz_j + sum4v(fz_ijv);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);
...
```

- Use your parallel programming skills to optimize performance

Performance increase comes from performing calculations on several particles in parallel

mitrion

Step 4: Optimize the Mitrion-C Code for Increased Performance

```
...
  (final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES_DIV>)
  {
     i = (e-1)*4;
     // read 4 particles into lists
     (x_il,y_il,z_il,mass_il) = foreach(v in <0..3>)
     {
         (x,y,z,mass) = read_particle(ram0, i+v);
     } (x,y,z,mass);
     x_iv    = reformat(x_il, [4]);
     y_iv    = reformat(y_il, [4]);
     z_iv    = reformat(z_il, [4]);
     mass_iv = reformat(mass_il, [4]);
    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in
                          fx,   fy,   fz   by j)
    {
       (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
        // match with 4 particles at a time
        (fx_ijv, fy_ijv, fz_ijv) = foreach (x_i, y_i, z_i,  mass_i in
                                            x_iv,y_iv,z_iv, mass_iv by v)
        {
            (fx_ij, fy_ij, fz_ij) = if(i != j)
            {
                Float dx = x_j - x_i;
                Float dy = y_j - y_i;
                Float dz = z_j - z_i;

                Float d = dx*dx + dy*dy + dz*dz;
                Float force = -0.000010 * mass_i * mass_j / d;

                Float x = force * dx;
                Float y = force * dy;
                Float z = force * dz;
            } (x,y,z)
            else
            { zero = 0;} (zero,zero,zero);
        } (fx_ij, fy_ij, fz_ij);
        new_fx_j = fx_j + sum4v(fx_ijv);
        new_fy_j = fy_j + sum4v(fy_ijv);
        new_fz_j = fz_j + sum4v(fz_ijv);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);
...
```

- Use your parallel programming skills to optimize performance
- ~~**Myth #2: Application development is complex and time-consuming**~~

mitrion

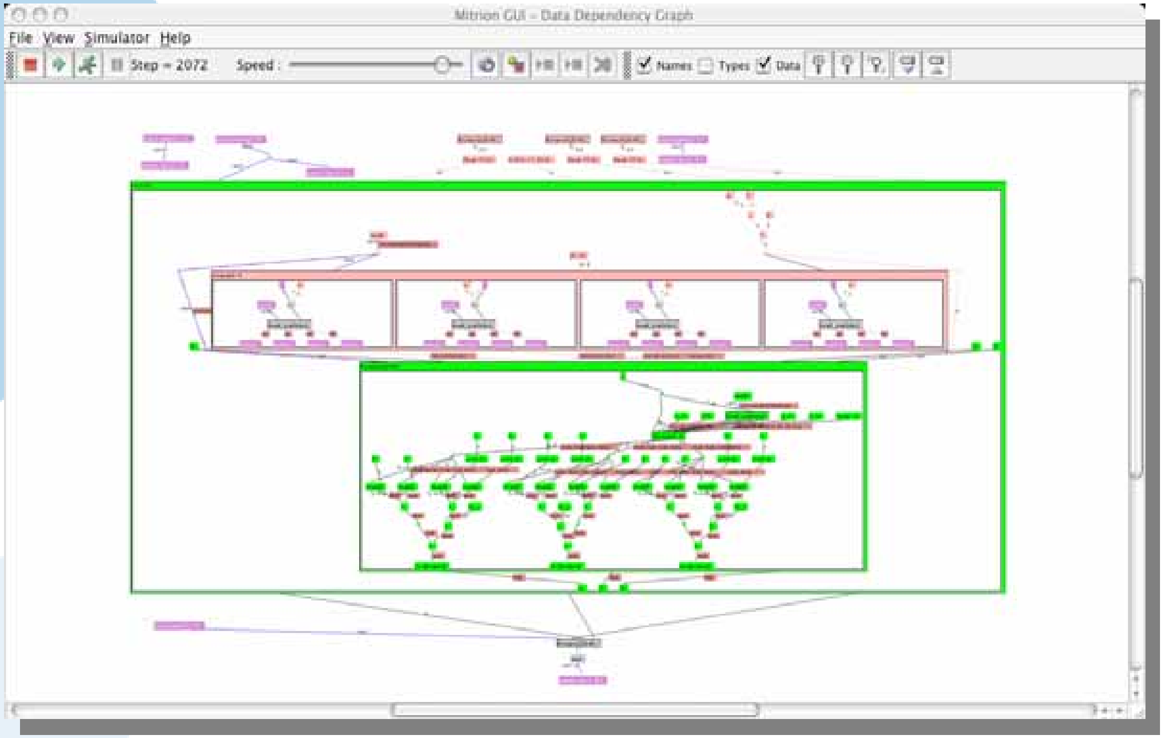
Myth #3:

A big initial investment is required in FPGA computers and EDA tool seats

mitrion

The Software to Get Started is All Free

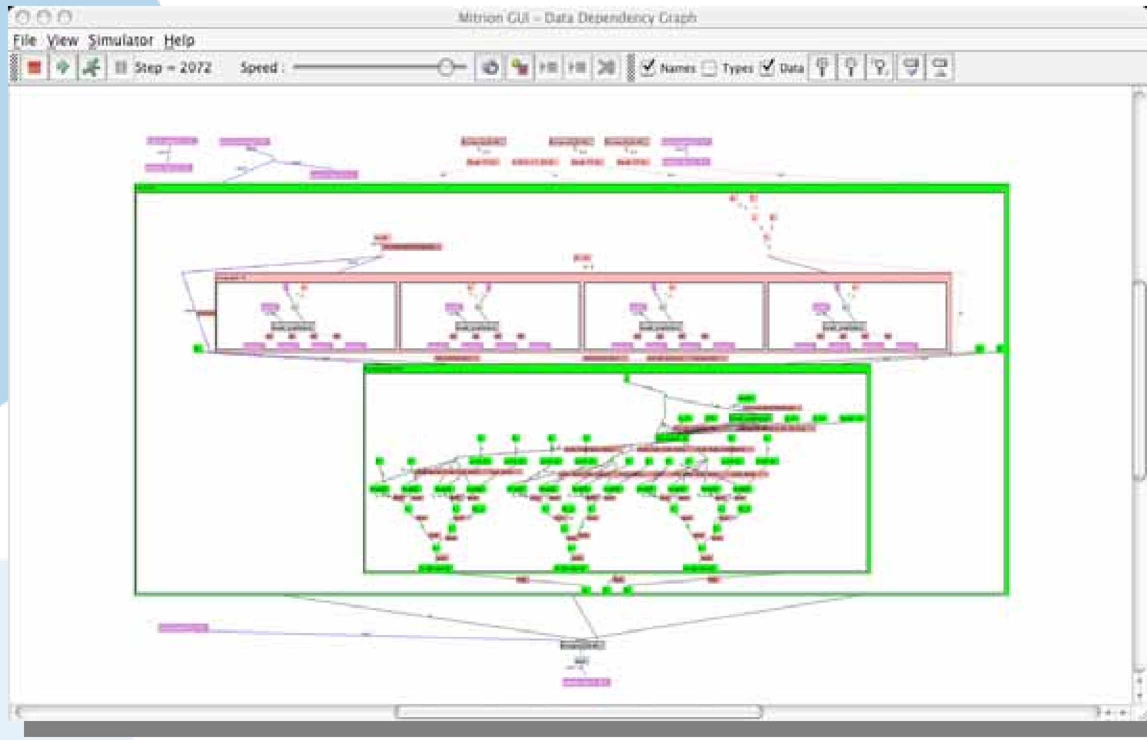
- Mitrion compiler and simulator available on request
- Java based environment runs on Linux, Windows, Mac
- Allows to compile, simulate, predict performance
- Mitrion processor needed to actually run



mitrion

The Software to Get Started is All Free

- Mitrion compiler and simulator available on request
- Java based environment runs on Linux, Windows, Mac
- Allows to compile, simulate, predict performance
- Mitrion processor needed to actually run



- **Myth #3: A big initial investment is needed in FPGA computers and EDA tool seats**

mitrion

Myth #4:

Lack of portability across FPGA generations and FPGA computers

mitrion

Step 5:
Let's Re-Write for Virtex 4

```
...
  (final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES_DIV>)
  {
     i = (e-1)*4;
     // read 4 particles into lists
     (x_il,y_il,z_il,mass_il) = foreach(v in <0..3>)
     {
         (x,y,z,mass) = read_particle(ram0, i+v);
     } (x,y,z,mass);
     x_iv    = reformat(x_il, [4]);
     y_iv    = reformat(y_il, [4]);
     z_iv    = reformat(z_il, [4]);
     mass_iv = reformat(mass_il, [4]);
    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in
                          fx,    fy,    fz    by j)
    {
      (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
       // match with 4 particles at a time
       (fx_ijv, fy_ijv, fz_ijv) = foreach (x_i, y_i, z_i,  mass_i in
                                   x_iv,y_iv,z_iv, mass_iv by v)
       {
           (fx_ij, fy_ij, fz_ij) = if(i != j)
           {
               Float dx = x_j - x_i;
               Float dy = y_j - y_i;
               Float dz = z_j - z_i;

               Float d = dx*dx + dy*dy + dz*dz;
               Float force = -0.000010 * mass_i * mass_j / d;

               Float x = force * dx;
               Float y = force * dy;
               Float z = force * dz;
           } (x,y,z)
           else
           { zero = 0;} (zero,zero,zero);
       } (fx_ij, fy_ij, fz_ij);
       new_fx_j = fx_j + sum4v(fx_ijv);
       new_fy_j = fy_j + sum4v(fy_ijv);
       new_fz_j = fz_j + sum4v(fz_ijv);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);
...
```

- Existing code will run on Virtex 4 as is, if recompiled for the new platform.

mitrion

Step 5:
Let’s Re-Write for Virtex 4

```
...
  (final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES_DIV>)
  {
     i = (e-1)*8;
     // read 8 particles into lists
     (x_il,y_il,z_il,mass_il) = foreach(v in <0..7>)
     {
         (x,y,z,mass) = read_particle(ram0, i+v);
     } (x,y,z,mass);
     x_iv    = reformat(x_il, [8]);
     y_iv    = reformat(y_il, [8]);
     z_iv    = reformat(z_il, [8]);
     mass_iv = reformat(mass_il, [8]);
    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in
                          fx,   fy,   fz   by j)
    {
      (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
       // match with 8 particles at a time
       (fx_ijv, fy_ijv, fz_ijv) = foreach (x_i, y_i, z_i, mass_i in
                                  x_iv,y_iv,z_iv, mass_iv by v)
       {
           (fx_ij, fy_ij, fz_ij) = if(i != j)
           {
               Float dx = x_j - x_i;
               Float dy = y_j - y_i;
               Float dz = z_j - z_i;

               Float d = dx*dx + dy*dy + dz*dz;
               Float force = -0.000010 * mass_i * mass_j / d;

               Float x = force * dx;
               Float y = force * dy;
               Float z = force * dz;
           } (x,y,z)
           else
           { zero = 0;} (zero,zero,zero);
       } (fx_ij, fy_ij, fz_ij);
       new_fx_j = fx_j + sum8v(fx_ijv);
       new_fy_j = fy_j + sum8v(fy_ijv);
       new_fz_j = fz_j + sum8v(fz_ijv);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);
...
```

- Existing code will run on Virtex 4 as is, if recompiled for the new platform.

For Virtex 4, we can perform calculations on even more particles in parallel

mitrion

Step 5:
Let's Re-Write for Virtex 4

```
...
  (final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES_DIV>)
  {
     i = (e-1)*8;
     // read 8 particles into lists
     (x_il,y_il,z_il,mass_il) = foreach(v in <0..7>)
     {
         (x,y,z,mass) = read_particle(ram0, i+v);
     } (x,y,z,mass);
     x_iv    = reformat(x_il, [8]);
     y_iv    = reformat(y_il, [8]);
     z_iv    = reformat(z_il, [8]);
     mass_iv = reformat(mass_il, [8]);
    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in
                          fx,   fy,   fz   by j)
    {
      (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
       // match with 8 particles at a time
       (fx_ijv, fy_ijv, fz_ijv) = foreach (x_i, y_i, z_i,  mass_i in
                                           x_iv,y_iv,z_iv, mass_iv by v)
       {
           (fx_ij, fy_ij, fz_ij) = if(i != j)
           {
               Float dx = x_j - x_i;
               Float dy = y_j - y_i;
               Float dz = z_j - z_i;

               Float d = dx*dx + dy*dy + dz*dz;
               Float force = -0.000010 * mass_i * mass_j / d;

               Float x = force * dx;
               Float y = force * dy;
               Float z = force * dz;
           } (x,y,z)
           else
           { zero = 0;} (zero,zero,zero);
       } (fx_ij, fy_ij, fz_ij);
       new_fx_j = fx_j + sum8v(fx_ijv);
       new_fy_j = fy_j + sum8v(fy_ijv);
       new_fz_j = fz_j + sum8v(fz_ijv);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);
...
```

- Existing code will run on Virtex 4 as is, if recompiled for the new platform.

Step 5: Let's Re-Write for Virtex 4

```
...
  (final_fx, final_fy, final_fz) = for(e in <1 ..  PARTICLES_DIV>)
  {
     i = (e-1)*8;
     // read 8 particles into lists
     (x_il,y_il,z_il,mass_il) = foreach(v in <0..7>)
     {
         (x,y,z,mass) = read_particle(ram0, i+v);
     } (x,y,z,mass);
     x_iv    = reformat(x_il, [8]);
     y_iv    = reformat(y_il, [8]);
     z_iv    = reformat(z_il, [8]);
     mass_iv = reformat(mass_il, [8]);
    (fx, fy, fz)= foreach(fx_j, fy_j, fz_j in
                          fx,   fy,   fz   by j)
    {
      (x_j, y_j, z_j, mass_j) = read_particle(ram0, j);
       // match with 8 particles at a time
       (fx_ijv, fy_ijv, fz_ijv) = foreach (x_i, y_i, z_i,  mass_i in
                                           x_iv,y_iv,z_iv, mass_iv by v)
       {
           (fx_ij, fy_ij, fz_ij) = if(i != j)
           {
               Float dx = x_j - x_i;
               Float dy = y_j - y_i;
               Float dz = z_j - z_i;

               Float d = dx*dx + dy*dy + dz*dz;
               Float force = -0.000010 * mass_i * mass_j / d;

               Float x = force * dx;
               Float y = force * dy;
               Float z = force * dz;
           } (x,y,z)
           else
           { zero = 0;} (zero,zero,zero);
       } (fx_ij, fy_ij, fz_ij);
       new_fx_j = fx_j + sum8v(fx_ijv);
       new_fy_j = fy_j + sum8v(fy_ijv);
       new_fz_j = fz_j + sum8v(fz_ijv);
    } (new_fx_j, new_fy_j, new_fz_j);
  } (fx, fy, fz);
...
```

- Existing code will run on Virtex 4 as is, if recompiled for the new platform.
- ~~**Myth #4: Lack of portability across FPGA generations and FPGA computers**~~

mitrion

~~The Obstacles~~ Myths

1. Electrical Engineering skills are necessary to program FPGAs
2. Application development is complex and time-consuming
3. A big initial investment is required in FPGA computers and EDA tool seats
4. Lack of portability across FPGA generations and FPGA computers

mitrion

Conclusion

- The promises of FPGA Supercomuting are real for many applications
- What used to be obstacles using older technology are now - myths
- It’s easy to evaluate FPGA feasibility, and to build and maintain HPC production environments using the Mitrion platform

mitrion

Thank you!

anders.dellson@mitrionics.com

www.mitrionics.com

mitrion

BACK-UP SLIDES

FPGAs – Fast or Slow?

- Just an empty re-configurable silicon surface
- 1,000 times *slower* than fixed silicon at the same process technology (90nm):
  - ~10 times slower clock frequency
  - ~100 times larger area used per gate
- But, 10-100 times *faster* compared to CPUs

mitrion

Processor Architecture: A Cluster-on-a-Chip

- Not Von Neumann architecture
- Processor architecture resembles a cluster
- Very Fine-Grain Parallelism
  - Normal clusters run a block of code on each PE
  - Mitrion runs a single instruction on each PE
  - Each PE adapted to optimally run its instruction
- Network topology specific to algorithm
- No Instruction Stream, instead Data Stream

The Mitrion-C Language

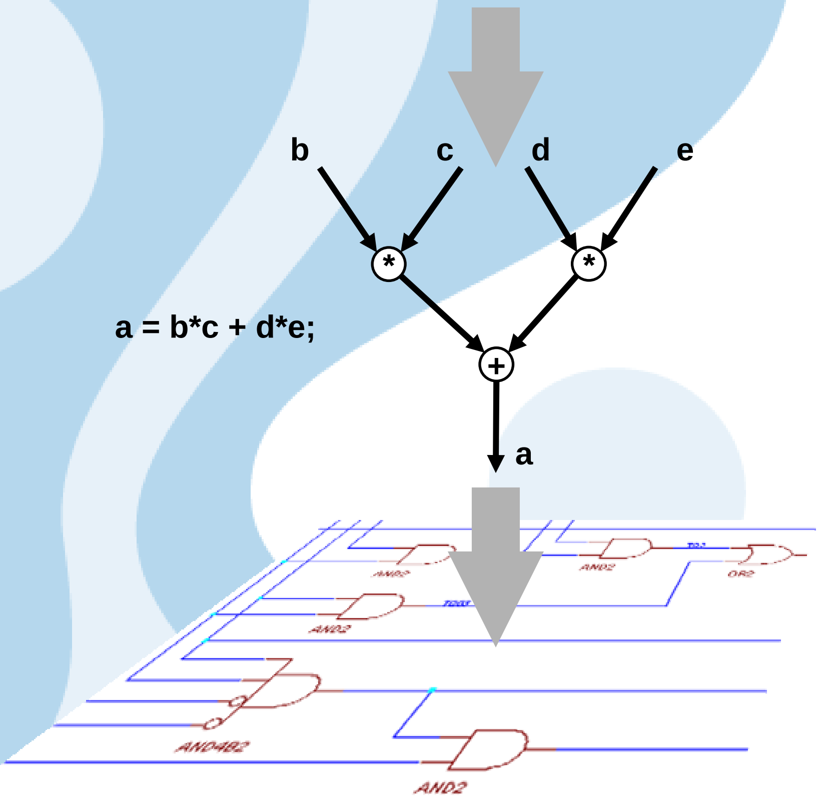
- The Mitrion Processor needs a fully parallel programming language
  - Languages with vector parallel extensions or simple parallel instructions not sufficient
- Main considerations
  - High parallelism
  - High programmability
  - No hardware design considerations

The Mitrion Virtual Processor

```
int:48<30> main()
{
  int:48 prev  = 1;
  int:48 fib   = 1;

  int:48<30> fibonnacci = for(i in <1..30>)
  {
    fib  = fib+prev;
    prev = fib;
  } <>fib;

} fibonnacci;
```



- A new processor architecture specifically for FPGAs

Architecture design goal:

- High silicon utilization
- Take advantage of FPGA re-configurability

Goal achieved by:

- Allow processor to be massively parallel
- Allow processor to be fully adapted to algorithm

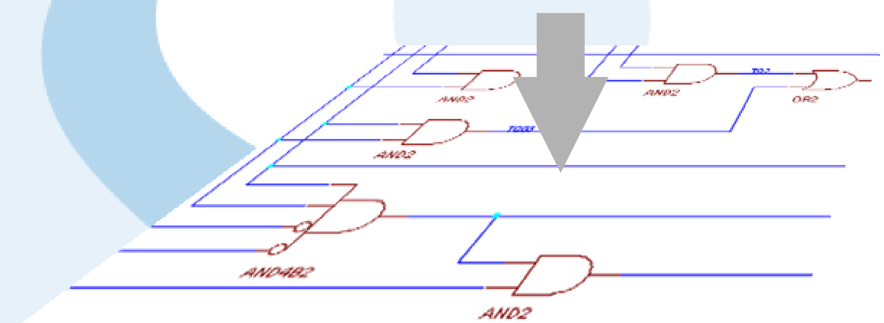
The Challenge:
Too Large Semantic Gap

```
int:48<30> main()
{
  int:48 prev  = 1;
  int:48 fib   = 1;

  int:48<30> fibonnacci = for(i in <1..30>)
  {
    fib  = fib+prev;
    prev = fib;
  } <>fib;

} fibonnacci;
```

?



Software:

Instruction stream for a processor

Hardware:

Transistors and wires

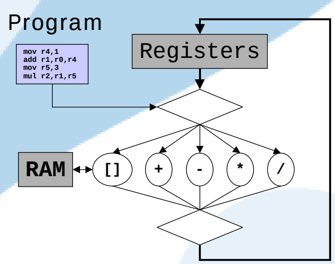
mitrion

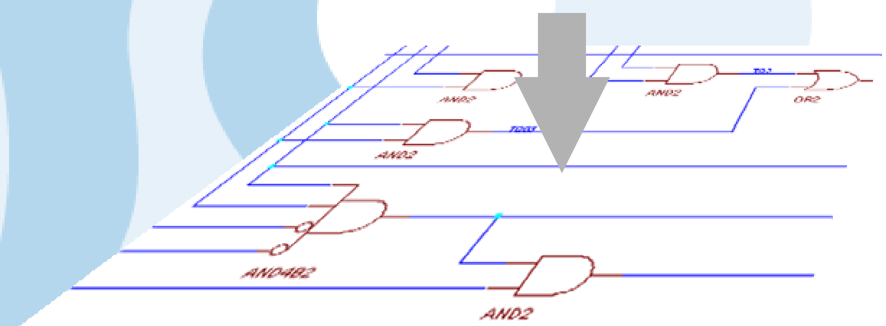
The von Neumann Architecture

```
int:48<30> main()
{
  int:48 prev  = 1;
  int:48 fib   = 1;

  int:48<30> fibonnacci = for(i in <1..30>)
  {
    fib  = fib+prev;
    prev = fib;
  } <>fib;

} fibonnacci;
```





The traditional von Neumann processor is a state machine, operating instructions one at a time that are read from RAM memory.

- \+ Easily programmable
- \+ Executes programs of any size
- – Single instruction stream gives very low parallelism
- – Low silicon utilization
- ➢ Needs very high clock frequency

mitrion

Step 2: Replace With Function Call to FPGA

```
nbody(float *x,   // input vectors
      float *y,
      float *z,
      float *mass,

      float *fx,  // output vectors
      float *fy,
      float *fz)
{
    int i;

    // Store positions and masses in FPGA RAM banks
    for( i = 0; i<PARTICLES; i++)
    {
        int off = i*4;
        ram[off+0] = x[i];
        ram[off+1] = y[i];
        ram[off+2] = z[i];
        ram[off+3] = mass[i];
    }

    // Start the Mitrion Virtual Processor
    mitrion_processor_run(p);
    // The run function is asynchronous, so we have to wait
    // explicitly. This call blocks until the MVP has finished.
    mitrion_processor_wait(p);


    // Read results back from FPGA RAMs
    for( i = 0; i<PARTICLES; i++)
    {
        int off = i*4;
        fx[i] = result_ram[off+0];
        fy[i] = result_ram[off+1];
        fz[i] = result_ram[off+2];
    }

}
```

- API calls are available to initialize and control the FPGA.
- Total effort: 2 hours

mitrion