Optimizing OpenCL™ on CPUs

Ofer Rosenberg
Visual Computing Software

* OpenCL is trademarks of Apple Inc. used by permission by Khronos
OpenCL and Heterogeneous computing

- OpenCL is a **Platform API** which supports a uniform programming environment across devices
  - Enables heterogeneous parallel computations
  - Unique in its ability to coordinate CPUs, GPUs, etc

- The goal of using OpenCL should be to make the best use of **all** the available resources (CPU’s, GPU’s) from within a **single** program:
  - One program that runs well (i.e. reasonably close to “hand-tuned” performance) on a heterogeneous mixture of processors.
  - Intel new generation of Processors: a new level of integration between CPU & GPU
Writing OpenCL for the CPU

- The challenge of unleashing the performance of modern CPU’s
  - Multi-Core / SMT
  - Vector Units (SSE ISA)

- OpenCL is a great framework to harness Intel CPUs
  - Intuitive, easy & maintainable
  - Unleash the performance of Intel CPUs
    - Multi Core
    - Utilize the vector units (SSE ISA)
    - Close to hand-tuned code!
  - Performance-portable code
    - Forward compatibility between CPU generations
    - Aspire compatibility between devices
OpenCL view of Core™ i7

Core™ i7 975
- 8 Compute Units
  - Quad Core + SMT
- 4/8/16 Processing Elements per Unit
  - 128bit XMM registers
  - Data type determines # of elements...
- (32K L1 + 256K L2) Per Core, 8M L3 Shared
  - Not part of OpenCL platform model, but useful 😊
Mapping OpenCL Data-Parallel Execution Model

• Implicit SIMD data parallelism (i.e. shader-style):
  – Write the kernel as a “scalar program”
  – Use vector data types sized naturally to the algorithm
  – Kernel automatically mapped to SIMD-compute-resources and cores by the compiler/runtime/hardware.

• Explicit SIMD data parallelism:
  – The kernel defines one stream of instructions
  – Parallelism from source-level wide vector types
    – Size vector types to match native HW width
  – Programmer hints on the vector data type using attributes
    – vec_type_hint(typen)
Example: N-body Simulation

Given $N$ bodies with an initial position $x_i$ and velocity $v_i$ for, the force $f_{ij}$ on body $i$ caused by body $j$ is given by following (G is gravity):

$$f_{ij} = G \frac{m_i m_j}{\|r_{ij}\|^2} \cdot \frac{r_{ij}}{||r_{ij}||}, \quad F_i = \sum_{1 \leq j \leq N \atop j \neq i} f_{ij}$$

where $m_i$ and $m_j$ are the masses of bodies $i$ and $j$, respectively; $r_{ij} = x_j - x_i$

The acceleration is computed as $a_i = F_i/m_i$
void nBodyFunc(
    const float *input_position_x, const float *input_position_y, const float *input_position_z,
    const float *input_velocity_x, const float *input_velocity_y, const float *input_velocity_z,
    float *output_position_x, float *output_position_y, float *output_position_z,
    float *output_velocity_x, float *output_velocity_y, float *output_velocity_z,
    const float *mass, int body_count, float softening_squared, float time_delta )
{
    for (int i = 0; i < body_count; i++)
    {
        float position_x = (float)(input_position_x[i]);
        float position_y = (float)(input_position_y[i]);
        float position_z = (float)(input_position_z[i]);
        float acc_x = 0, acc_y = 0, acc_z = 0;

        for (int j = 0; j < body_count; j++)
        {
            float dx = input_position_x[j] - position_x;
            float dy = input_position_y[j] - position_y;
            float dz = input_position_z[j] - position_z;

            float distance_squared = dx * dx + dy * dy + dz * dz + softening_squared;
            float inverse_distance = rsqrt(distance_squared);
            float s = (mass[j] * inverse_distance) * (inverse_distance * inverse_distance);

            acc_x += dx * s; acc_y += dy * s; acc_z += dz * s;
        }
        output_velocity_x[i] = input_velocity_x[i] + acc_x * time_delta;
        output_velocity_y[i] = input_velocity_y[i] + acc_y * time_delta;
        output_velocity_z[i] = input_velocity_z[i] + acc_z * time_delta;

        output_position_x[i] = input_position_x[i] + input_velocity_x[i] * time_delta + acc_x * time_delta * time_delta/2;
        output_position_y[i] = input_position_y[i] + input_velocity_y[i] * time_delta + acc_y * time_delta * time_delta/2;
        output_position_z[i] = input_position_z[i] + input_velocity_z[i] * time_delta + acc_z * time_delta * time_delta/2;
    }
}
NBody – from C to CL

```c
void nBodyFunc(
    const float *input_position_x, const float *input_position_y, const float *input_position_z,
    const float *input_velocity_x, const float *input_velocity_y, const float *input_velocity_z,
    float *output_position_x, float *output_position_y, float *output_position_z,
    float *output_velocity_x, float *output_velocity_y, float *output_velocity_z,
    const float *mass, int body_count, float softening_squared, float time_delta )
{
    for (int i = 0; i < body_count; i++)
    {
        float position_x = (float)(input_position_x[i]);
        float position_y = (float)(input_position_y[i]);
        float position_z = (float)(input_position_z[i]);
        float acc_x = 0, acc_y = 0, acc_z = 0;

        for (int j = 0; j < body_count; j++)
        {
            float dx = input_position_x[j] - position_x;
            float dy = input_position_y[j] - position_y;
            float dz = input_position_z[j] - position_z;

            float distance_squared = dx * dx + dy * dy + dz * dz + softening_squared;
            float inverse_distance = rsqrt(distance_squared);
            float s = (mass[j] * inverse_distance) * (inverse_distance * inverse_distance);

            acc_x += dx * s; acc_y += dy * s; acc_z += dz * s;
        }

        output_velocity_x[i] = input_velocity_x[i] + acc_x * time_delta;
        output_velocity_y[i] = input_velocity_y[i] + acc_y * time_delta;
        output_velocity_z[i] = input_velocity_z[i] + acc_z * time_delta;

        output_position_x[i] = input_position_x[i] + input_velocity_x[i] * time_delta + acc_x * time_delta * time_delta/2;
        output_position_y[i] = input_position_y[i] + input_velocity_y[i] * time_delta + acc_y * time_delta * time_delta/2;
        output_position_z[i] = input_position_z[i] + input_velocity_z[i] * time_delta + acc_z * time_delta * time_delta/2;
    }
}
```

External loop: calculate new position & speed for all bodies

Internal loop: calculate influence of all the bodies on \([i]\)

Calculate \([i]\)’s new position & speed
```
__kernel void nBodyKernel(
    const __global float *input_position_x, const __global float *input_position_y, const __global float *input_position_z,
    const __global float *input_velocity_x, const __global float *input_velocity_y, const __global float *input_velocity_z,
    __global float *output_position_x, __global float *output_position_y, __global float *output_position_z,
    __global float *output_velocity_x, __global float *output_velocity_y, __global float *output_velocity_z,
    const __global float *mass, int body_count, float softening_squared, float time_delta )
{
    int index = get_global_id(0);

    float position_x = (float)(input_position_x[index]);
    float position_y = (float)(input_position_y[index]);
    float position_z = (float)(input_position_z[index]);
    float acc_x = 0, acc_y = 0, acc_z = 0;

    for (int j = 0; j < body_count; j++)
    {
        float dx = input_position_x[j] - position_x;
        float dy = input_position_y[j] - position_y;
        float dz = input_position_z[j] - position_z;

        float distance_squared = dx * dx + dy * dy + dz * dz + softening_squared;
        float inverse_distance = rsqrt(distance_squared);
        float s = (mass[j] * inverse_distance) * (inverse_distance * inverse_distance);

        acc_x += dx * s; acc_y += dy * s; acc_z += dz * s;
    }

    output_velocity_x[index] = input_velocity_x[index] + acc_x * time_delta;
    output_velocity_y[index] = input_velocity_y[index] + acc_y * time_delta;
    output_velocity_z[index] = input_velocity_z[index] + acc_z * time_delta;

    output_position_x[index] = input_position_x[index] + input_velocity_x[index] * time_delta + acc_x * time_delta * time_delta/2;
    output_position_y[index] = input_position_y[index] + input_velocity_y[index] * time_delta + acc_y * time_delta * time_delta/2;
    output_position_z[index] = input_position_z[index] + input_velocity_z[index] * time_delta + acc_z * time_delta * time_delta/2;
}
```
NBody – from C to CL

```c
__kernel void nBodyKernel(
    const __global float *input_position_x, const __global float *input_position_y, const __global float *input_position_z,
    const __global float *input_velocity_x, const __global float *input_velocity_y, const __global float *input_velocity_z,
    __global float *output_position_x, __global float *output_position_y, __global float *output_position_z,
    __global float *output_velocity_x, __global float *output_velocity_y, __global float *output_velocity_z,
    const __global float *mass, int body_count, float softening_squared, float time_delta)
{
    int index = get_global_id(0);

    float position_x = (float)(input_position_x[index]);
    float position_y = (float)(input_position_y[index]);
    float position_z = (float)(input_position_z[index]);
    float acc_x = 0, acc_y = 0, acc_z = 0;

    for (int j = 0; j < body_count; j++)
    {
        float dx = input_position_x[j] - position_x;
        float dy = input_position_y[j] - position_y;
        float dz = input_position_z[j] - position_z;

        float distance_squared = dx * dx + dy * dy + dz * dz + softening_squared;
        float inverse_distance = rsqrt(distance_squared);
        float s = (mass[j] * inverse_distance) * (inverse_distance * inverse_distance);

        acc_x += dx * s; acc_y += dy * s; acc_z += dz * s;
    }

    output_velocity_x[index] = input_velocity_x[index] + acc_x * time_delta;
    output_velocity_y[index] = input_velocity_y[index] + acc_y * time_delta;
    output_velocity_z[index] = input_velocity_z[index] + acc_z * time_delta;

    output_position_x[index] = input_position_x[index] + input_velocity_x[index] * time_delta + acc_x * time_delta * time_delta/2;
    output_position_y[index] = input_position_y[index] + input_velocity_y[index] * time_delta + acc_y * time_delta * time_delta/2;
    output_position_z[index] = input_position_z[index] + input_velocity_z[index] * time_delta + acc_z * time_delta * time_delta/2;
}
```

What do the developer gains from converting to OpenCL C?
NBody Performance

Results from Intel’s internal OpenCL implementation: *

* Results measured on Core™ i7 975, 3.3 GHz, 6GB DDR3
  Results depends on the algorithm/code running
NBody Performance

Results from Intel’s internal OpenCL implementation: *

- Porting C to CL - Implicit Data Parallelism
  - “shader-style” code
  - Benefit from multi-core/SMT

* Results measured on Core™ i7 975, 3.3 GHz, 6GB DDR3
Results depends on the algorithm/code running
NBody Performance

Results from Intel’s internal OpenCL implementation: *

- Porting C to CL - Implicit Data Parallelism
  - “shader-style” code
  - Benefit from multi-core/SMT

- Implicit Data Parallelism with Vectorization
  - Intel’s internal implementation
  - Cross-workitem Vectorization/Packing
  - Benefit from SSE (128bit registers)

* Results measured on Core™ i7 975, 3.3 GHz, 6GB DDR3
Results depends on the algorithm/code running
NBody Performance

Results from Intel’s internal OpenCL implementation: *

- Porting C to CL - Implicit Data Parallelism
  - “shader-style” code
  - Benefit from multi-core/SMT

- Implicit Data Parallelism with Vectorization
  - Intel’s internal implementation
  - Cross-workitem Vectorization/Packing
  - Benefit from SSE (128bit registers)

- Explicit Data-Parallelism
  - Hand tuned OpenCL C code

* Results measured on Core™ i7 975, 3.3 GHz, 6GB DDR3
Results depends on the algorithm/code running

OpenCL Explicit version is x25 faster than Naïve C *
Explicit version is only 14% slower than highly optimized code *
### Implicit Data Parallelism on the CPU

**On Intel internal Implementation**

- One workitem runs on a single SSE lane
- Workitems are packed to SSE registers as part of **Intel OpenCL Compilation process**
  - One code stream is mapped to one SSE lane
  - All operations/calls are vectorized
- Vector data types inside the kernel code are scalarized and mapped to a single SSE lane
- The Vectorizer generates a Workgroup
  - Further workgroup level optimizations
- Workgroup is executed on a compute unit (HW Thread)

- Kernel is executed over an N-D Range, which is divided to workgroups
- Several Workgroups run concurrently on all compute unit (HW threads)
Overview of the Vectorization Stage

```c
__kernel void program(float4* pos, int numBodies, float deltaTime)
{
    float myPos = gid;
    float refPos = numBodies + deltaTime;
    float4 r = pos[refPos - myPos];
    float distSqr = r.x * r.x + r.y * r.y + r.z * r.z;
    float invDist = sqrt(distSqr + epsSqr);
    float invDistCube = invDist * invDist * invDist;
    float4 acc = invDistCube * r;
    float4 oldVel = vel(gid);
    float newPos = myPos.w;
}
```
Overview of the Vectorization Stage

__kernel void program(float * pos, int numBodies, float deltaTime)
{
    float myPos = gid;
    float refPos = numBodies + deltaTime;
    float4 r = pos[refPos - myPos];
    float distSqr = r.x * r.x + r.y * r.y + r.z * r.z;
    float invDist = sqrt(distSqr + epsSqr);
    float invDistCube = invDist * invDist * invDist;
    float4 acc = invDistCube * r;
    float4 oldVel = vel(gid);
    float newPos = myPos.w;
}

Multiple work items

Next: Visualize
Overview of the Vectorization Stage

```c
__kernel void program(float4* pos, int numBodies, float deltaTime) {
    float myPos = gid;
    float refPos = numBodies + deltaTime;
    float4 r = pos[refPos - myPos];
    float distSqr = r.x * r.x + r.y * r.y + r.z * r.z;
    float invDist = sqrt(distSqr + epsSqr);
    float invDistCube = invDist * invDist * invDist;
    float4 acc = invDistCube * r;
    float4 oldVel = vel(gid);
    float newPos = myPos.w;
}
```

Vector instructions

Graphic visualization…

Next: Scalarize
Overview of the Vectorization Stage

Scalarizing code…
Next: Vectorize
Overview of the Vectorization Stage

Vectorizing code…
Overview of the Vectorization Stage

Vectorizing code…
Overview of the Vectorization Stage

Vectorization enables developer to exploit the CPU Vector Units in Implicit Data Parallelism

Reduced number of invocations
Explicit Data Parallelism on the CPU

- Workitem is executed solely on a single compute unit (HW Thread)
- Vector operations are mapped to SSE instructions
- Vectors which are wider than the physical register are processed by interleaving

- Workgroup is executed on a single compute unit (HW Thread)
- Barrier & Fence built-ins impose some penalty (context saving)

- Kernel is executed over an N-D Range, which is divided to workgroups
- Several Workgroups run concurrently on all compute unit (HW threads)
Demo
Summary

Conclusions

• OpenCL provides the developer with the tools to unleash the performance of CPU’s
  – Multi-thread/SMT, Vector units (SSE)
  – Forward Compatibility

• OpenCL supports two forms of data parallelism, both map well to Intel Architectures

• Implicit Data Parallelism has the best chance of mapping onto a diverse range of hardware
  – Requires advanced compilation techniques

• Intel® OpenCL SDK “Alpha” software will be available by end of 2010 on whatif.intel.com
References

- **s09.idav.ucdavis.edu** for slides from a Siggraph2009 course titled “Beyond Programmable Shading”


Legal Disclaimer

- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.

- Intel may make changes to specifications and product descriptions at any time, without notice.

- All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

- Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

- Any code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel’s internal code names is at the sole risk of the user.

- Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

- Intel, Intel Inside, the Intel logo, and Intel Core are trademarks of Intel Corporation in the United States and other countries.

- OpenCL is trademarks of Apple Inc. used by permission by Khronos.

- *Other names and brands may be claimed as the property of others.

- Copyright © 2010 Intel Corporation. All rights reserved.