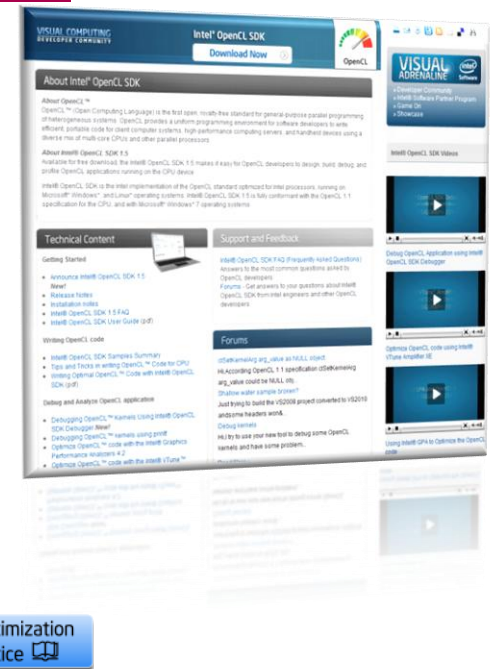


Intel OpenCL Team

- Responsible for Intel® OpenCL SDK for Intel® Architecture.
- Develop the LLVM-based OpenCL* compiler, debugger, etc.
- Enable future architectures in LLVM.
- The group is centered in Haifa, Israel.

Intel® OpenCL SDK 1.5

- Publicly Available Today
- Features:
 - Conformant with the OpenCL* 1.1 specification.
 - Supports 32/64-bit Microsoft Windows* operating systems and 64-bit Linux* operating systems.
 - Unique Implicit Vectorization Module.
- Available at: www.intel.com/software/opencl



Intel® OpenCL SDK Offline Compiler

- The Intel® OpenCL SDK Offline Compiler exposes LLVM-IR and enables users to examine the vectorized OpenCL* kernel.
- Tool [demo](#)

OpenCL Code

LLVM-IR

The screenshot displays the Intel OpenCL SDK Offline Compiler (v2.14.0) interface. The main window is divided into three panes:

- OpenCL Code:** Contains the source code for a kernel named 'GodRays.cl'. The code includes comments and C-style syntax for ray tracing calculations, such as 'Fix Decay by the step length' and 'Crop the ray by the image borders'.
- LLVM-IR:** Displays the LLVM Intermediate Representation of the OpenCL code. It shows declarations for global variables and functions, followed by a series of instructions (e.g., 'extractelement', 'add', 'icmp', 'select') that represent the compiled kernel logic.
- Build Log:** Shows the output of the compilation process. It indicates that the default instruction set architecture was used, the OpenCL CPU device was found, and the kernel was successfully built. The log also notes that the kernel was not vectorized.



OpenCL*

- A framework for developing data-parallel programs for multiple kinds of devices.
- An open standard by Khronos*.
- C-based language for compute shaders (called kernels):
 - Derived from ISO C99
 - Few restrictions, e.g. recursion, function pointers
 - Short vector types e.g., float4, short2, int16
 - Built-in functions: math (e.g., sin), geometric, common (e.g., min, clamp)

OpenCL* Kernels

Serial

```
void serial_mul(int n,  
               const float *a,  
               const float *b,  
               float *result){  
    int i;  
    for (i=0; i<n; i++)  
        result[i] = a[i] * b[i];  
}
```



Data Parallel


```
kernel void  
cl_mul(global const float *a,  
        global const float *b,  
        global float *result) {  
    int id = get_global_id(0);  
    result[id] = a[id] * b[id];  
}
```



Iterations – the “work” that needs to be done.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----

- The OpenCL runtime executes the kernel on each one of the work-items.
- A kernel processes a single work-item.



```
kernel void
cl_mul(global const float *a,
       global const float *b,
       global float *result) {

    int id = get_global_id(0);

    float k = sin(b[id]);

    result[id] = a[id] * k;
}
```

Iterations – the “work” that needs to be done.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----

```
kernel void
cl_mul(global const float *a,
global const float *b,
global float *result) {
    int id = get_global_id(0);
    float k = sin(b[id]);
    result[id] = a[id] * k;
}
```

```
kernel void
cl_mul(global const float *a,
global const float *b,
global float *result) {
    int id = get_global_id(0);
    float k = sin(b[id]);
    result[id] = a[id] * k;
}
```

```
kernel void
cl_mul(global const float *a,
global const float *b,
global float *result) {
    int id = get_global_id(0);
    float k = sin(b[id]);
    result[id] = a[id] * k;
}
```

```
kernel void
cl_mul(global const float *a,
global const float *b,
global float *result) {
    int id = get_global_id(0);
    float k = sin(b[id]);
    result[id] = a[id] * k;
}
```

```
kernel void
cl_mul(global const float *a,
global const float *b,
global float *result) {
    int id = get_global_id(0);
    float k = sin(b[id]);
    result[id] = a[id] * k;
}
```

```
kernel void
cl_mul(global const float *a,
global const float *b,
global float *result) {
    int id = get_global_id(0);
    float k = sin(b[id]);
    result[id] = a[id] * k;
}
```

```
kernel void
cl_mul(global const float *a,
global const float *b,
global float *result) {
    int id = get_global_id(0);
    float k = sin(b[id]);
    result[id] = a[id] * k;
}
```

```
kernel void
cl_mul(global const float *a,
global const float *b,
global float *result) {
    int id = get_global_id(0);
    float k = sin(b[id]);
    result[id] = a[id] * k;
}
```

- Vectorized kernels process multiple work-items at once.
- Less kernel invocations are needed.

```
kernel void
cl_mul(global const float *a,
      global const float *b,
      global float *result) {

    int id = get_global_id(0);

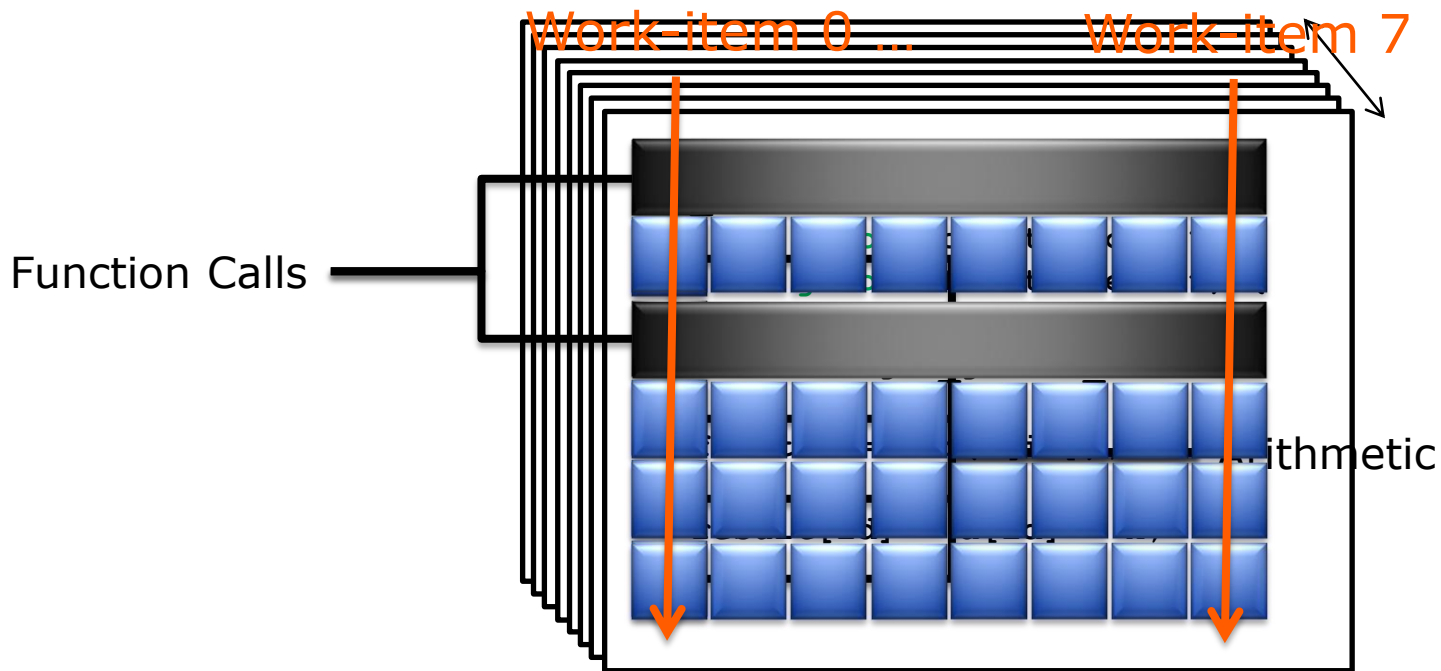
    float k = sin(b[id]);

    result[id] = a[id] * k;
}
```



What is Data-Parallel Vectorization ?

1. Input: LLVM-IR
2. Widen each instruction to make use of SSE4/AVX
3. Reduce the number of kernel invocations



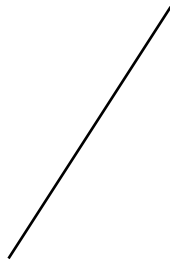
Vectorization in LLVM-IR

`%b = add i32 %a, %a`

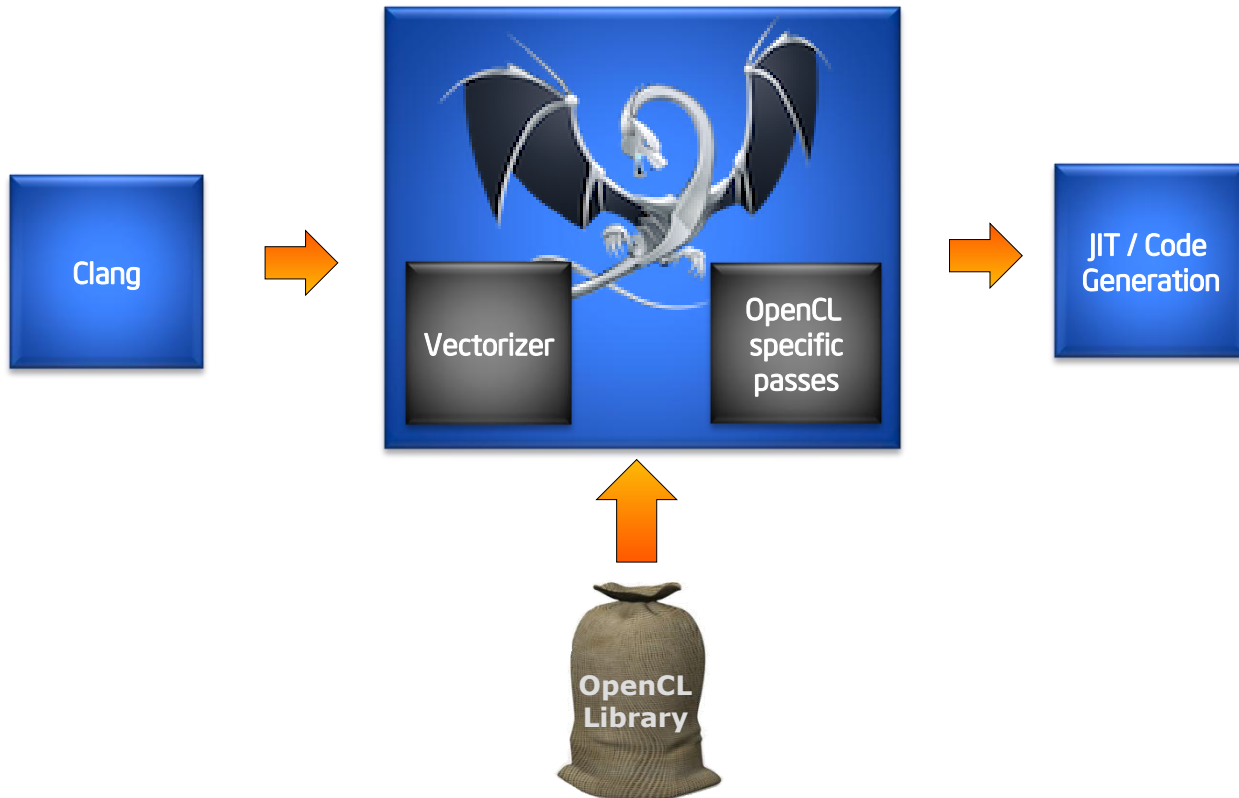
becomes

`%b = add <8 x i32> %a, %a`

Vectorization factor

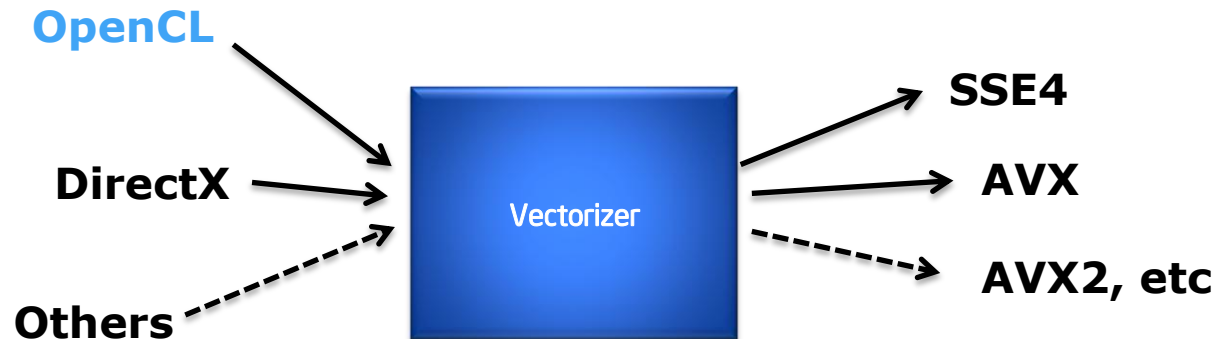


Intel® OpenCL SDK Structure



Vectorizer

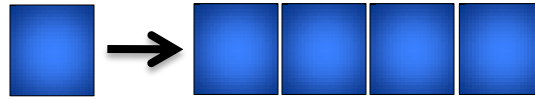
- Used in Intel® OpenCL SDK as well as other data-parallel products.
- Generates code for multiple Intel devices.
- A sequence of LLVM transformation and analysis passes.



Portable Vectorizer

- The vectorizer has multiple data-parallel “users”.
- Language dependent information is provided by an external analysis pass.
- For example:
 - In OpenCL* `get_global_id()` returns a consecutive index.
 - In Microsoft DirectX* API, ‘sync’ is a synchronization function.
 - List of built-in functions.
- LLVM allows high flexibility and enables vectorization that is language and platform independent.

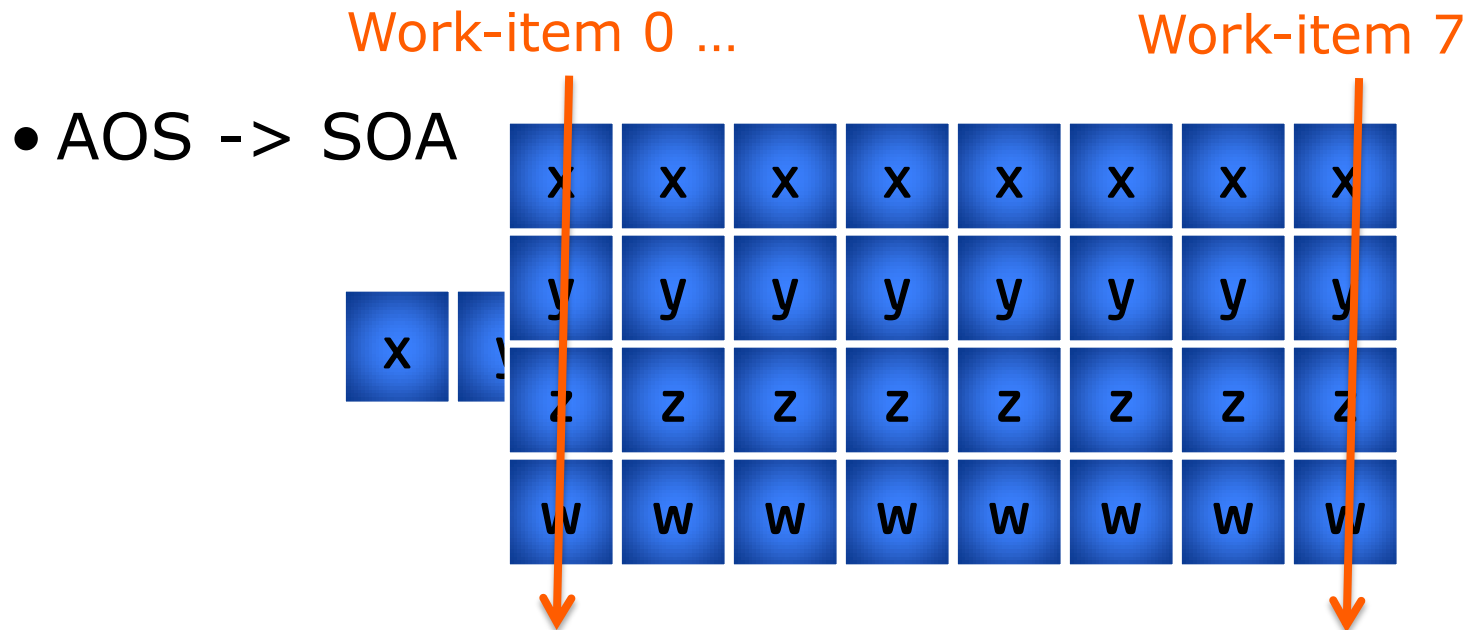
Packetization



- Widens a single element into a vector of elements.
- Many details are handled by the packetizer:
 - Widening of arithmetic scalar LLVM instructions to vector LLVM instructions.
 - Replacing function calls with wide versions.
 - For example: `sin(float)` -> `sin4(float4)`
 - Optimized widening of load/stores.

Packetization of 'Vector' Code

- OpenCL* supports vector data types and some users write vector code.
- We scalarize all incoming user vectors and re-vectorize to make use of the wide instruction set.



Predication

- When control-flow diverges, it is illegal to widen instructions.

```
__kernel void program(float* pos) {  
    int idx = get_global_id(0);  
    if (idx > 17) pos[idx] = 7;  
}
```



Not all
work-items
store.

- Need to use masks (predicates) to make all of the work-items execute the same control flow path.
- Need to handle diverging branches, loops, etc.

Vectorization Flow

```
__kernel void program(float* pos) {  
    int idx = get_global_id(0);  
    if (idx > 17) pos[idx] = 7;  
}
```

Predicate

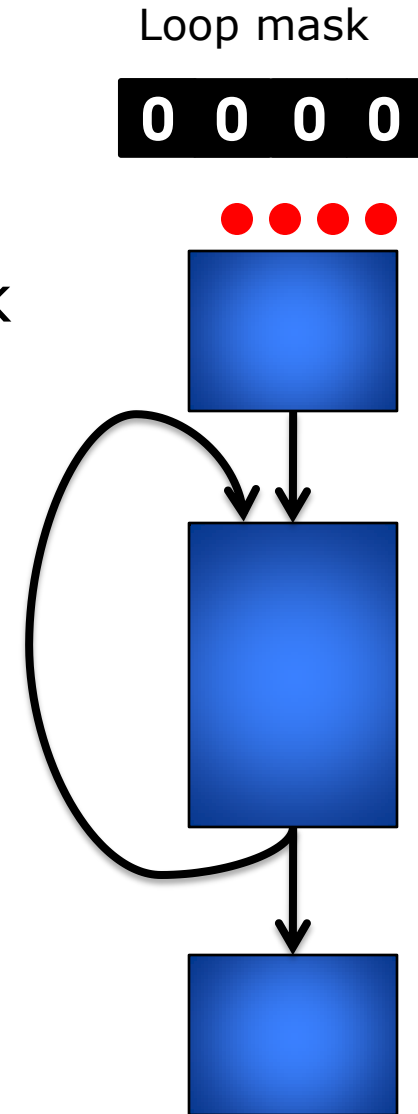
```
__kernel void program(float* pos) {  
    int idx = get_global_id(0);  
    bool p = (idx > 17);  
    masked_store(p, pos+idx, 7);  
}
```

Packetize

```
__kernel void program(float* pos) {  
    int4 idx = get_global_id4(0);  
    bool4 p = (idx > 17);  
    masked_store4(p, pos+idx, 7);  
}
```

Control Flow in 'while' Loops

- In loops, work-items enter at the same time, but exit at different times.
- Work items need to wait for the last work item to finish executing.
- Mask starts with all-one, and drops to zero for each work-item leaving the loop.

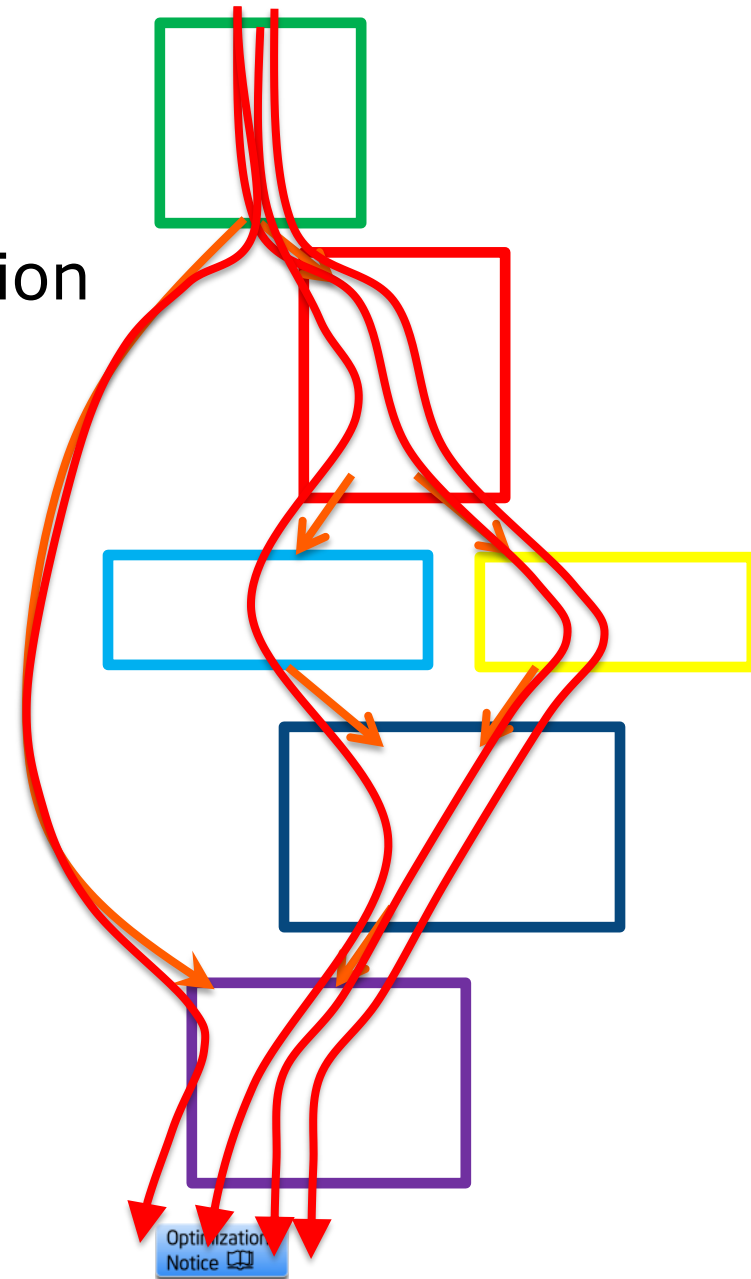
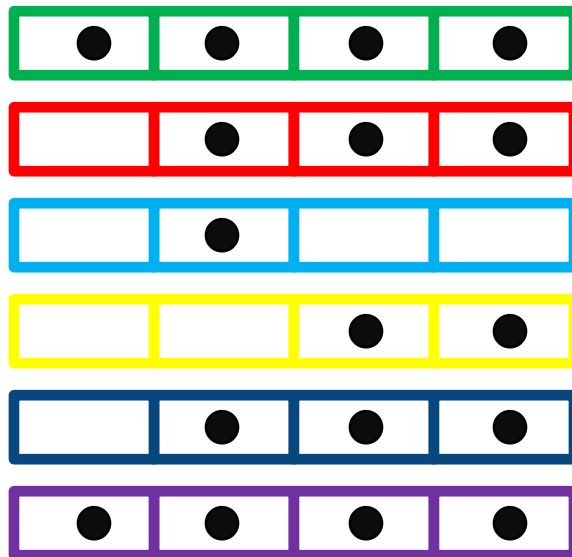


Masking of Functions and Memory Ops

- We mask: function calls, loads, stores, etc.
- LLVM does not support IR-level predication.
- Use Intrinsic functions:
 - Each architecture requires a different set of intrinsics.
 - Dedicated code to handle the widening of masked functions.
 - Dedicated code to implement unsupported memory ops, scatter/gather, etc.

Partial Vector Utilization

- Predication flattens the CF and execs both the 'then' and 'else'.
- Diverging CF reduces the utilization of vector instructions.
- Vectorization adds masking-overhead.

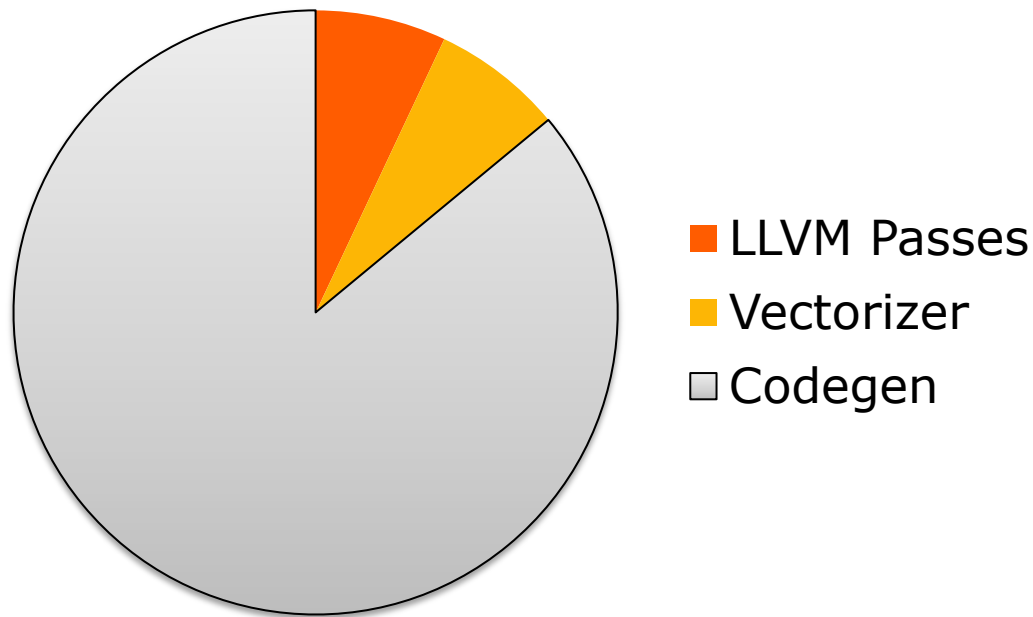


Vectorization Width Decision

- A heuristics pass is used to decide what is the ideal vectorization width (1, 4, 8, etc.)
- Perform static-analysis to collect different parameters of the program.
- We decide what vectorization-factor is best for the program:
 - Programs with low vector-utilizations do not get vectorized.
 - Integer-heavy kernels usually run better on vectorization factor of 4 because AVX is focused on floating-point types.
- Tuned using a large number of OpenCL* programs.

Typical Build Time

- The vectorizer is a fast pass.
- The vectorizer increases the codegen build time.



Vector-Select Codegen support

- Our vectorizer uses the vector-select instruction extensively.

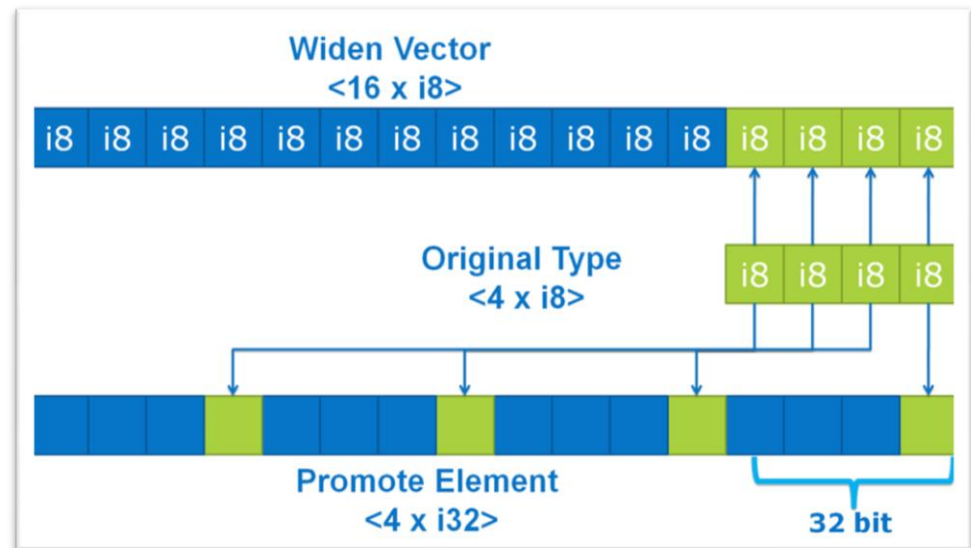
```
%C = fcmp oeq <8 x float> %A, %B
```

```
%V = select <8 x i1> %C, <8 x float> %A, <8 x float> %B
```

- Initially, we implemented a pass to lower vector-compare and vector-select sequences to x86 (AVX and SSE) intrinsics.
- Worked with the community on implementing codegen support for vector-select.

Vector Select and Type-Legalization

- Added a new kind of type-legalization:
 “vector-promotion”
- Added support for vector-select on top:
 - Added x86 optimized blend sequences for AVX, SSE4
 - Implemented using XOR, AND, for other targets.
- Enabled in ToT.



LLVM Wishlist

- Predicated instructions:
 - Needed for vector architectures, which will likely to dominate future processors
 - Need IR changes
- Improved Microsoft Windows* operating systems support:
 - Calling Conventions, MC, etc.
- Backward compatibility of the IR:
 - Use LLVM-IR as a cross-platform IR for OpenCL
 - Ease migration between versions of LLVM

Intel OpenCL Team and the Community

- We've contributed bug fixes and features to LLVM.
- We've developed our own AVX support on top of LLVM 2.8.
 - Started contributing our improvements.
- We plan to work on AVX2 together with the community.
- We intend to work with the LLVM community on improving LLVM for Intel Platforms.



Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Cilk, Core Inside, FlashFile, i960, InstantIP, Intel, the Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

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

Copyright © 2011. Intel Corporation.

<http://intel.com/software/products>