Compilation from and to stream languages

Alexandre Isoard

supervised by Alain Darte
Compsys Team, LIP, ENS de Lyon

Ph.D Seminar
5 March 2015
Table of contents

1 Introduction
   - Who? When? Where?
   - What? Why?

2 Kernel Offloading
   - Difficulties
   - Polynomial Product

3 Polyhedral Model
   - Computability
   - Limits
   - Extensions
Compsys Team

Permanents:

- **Alain Darte**: Memory folding, Software pipelining, ...
- Paul Feautrier: Extending the polyhedral model, ...
- Christophe Alias: High Level Synthesis, FPGA, ...
- Laure Gonnord: Abstract interpretation, ...
- Yuki Tomofumi: User-Compiler interaction, ...

Ph.D students:

- Guillaume Iooss: Semantic Tiling
- Alexandre Isoard: Compilation to and from stream languages
- Maroua Maalej: Low-cost static analysis for efficient compilers

Interns:

- Adilla Susungi: GP-GPU stuff
Parallel programming is **HARD**!

- Communication
- Synchronization
- Data-locality
- Pipelining

*Threads* are hard to do right:

- Light/system threads? Dynamic/static scheduling?
- Size of local buffers? Amount of data transfers?

Lets the compiler do some\(^1\) of it for you!

---

\(^1\)intrinsically not computable
Kernel Offloading

Let's say you want to compute a *really big* polynomial product:

\[ C(x) = \sum_{k \in [0,n)} c_k x^k \quad c_k = \sum_{i+j=k \atop i, j \in [0,n)} a_i b_j \]
Kernel Offloading

Let’s say you want to compute a *really big* polynomial product:

\[ C(x) = \sum_{k \in [0,n)} c_k x^k \quad c_k = \sum_{i+j=k \atop i,j \in [0,n)} a_i b_j \]

So you write this:

```c
for (int k=0; k<2*n-1; k++)
S0: C[k] = 0;
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
S1: C[i+j] += A[i]*B[j];
```
Kernel Offloading

Lets say you want to compute a really big polynomial product:

\[ C(x) = \sum_{k \in [0,n)} c_k x^k \quad c_k = \sum_{i+j=k, i,j \in [0,n)} a_i b_j \]

So you write this:

```c
for (int k=0; k<2*n-1; k++)
  S0: C[k] = 0;
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
    S1: C[i+j] += A[i]*B[j];
```

I have this really powerfull GPU, why not use it?
Kernel Offloading

- Host
- CPU
- Global Memory
- Local Memory
- Accelerator (FPGA/GPU/MPPA/...)

Memory Access:
- Slow: Host to Global Memory, Global Memory to Local Memory
- Fast: Local Memory to Accelerator, Accelerator to Local Memory
Pipelining

Problem: High latency channel, low throughput

Solution:
- Pre-fetch: Send next computation input data while computing
- Reuse: Locally keep reused result (required if dependencies)
- Coalescing: Send continuous chunk of data (maximize throughput)
Pipelining

Problem: High latency channel, low throughput

Solution:

- Pre-fetch: Send next computation input data while computing
- Reuse: Locally keep reused result (required if dependencies)
- Coalescing: Send continuous chunk of data (maximize throughput)

Improving data locality improve all of them.
Perform computations by blocks;
Exploit data reuse;
Use pipelining/prefetching;
Reduce and coalesce communications (burst).
Reads, writes, schedule

Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k = 0; k < 2*n - 1; k++)
    S0: C[k] = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        S1: C[i+j] += A[i]*B[j];
```
Reads, writes, schedule

Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k = 0; k < 2*n - 1; k++)
S0: C[k] = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        S1: C[i+j] += A[i]*B[j];
```
**Reads, writes, schedule**

Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k=0; k<2*n-1; k++)
    S0: C[k] = 0;
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        S1: C[i+j] += A[i]*B[j];
```

```c++
for (int k=0; k<2*n-1; k++)
    S0: C[k] = 0;
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        S1: C[i+j] += A[i]*B[j];
```
Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for(int k=0; k<2*n-1; k++)
    S0: C[k] = 0;
for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
        S1: C[i+j] += A[i]*B[j];
```
Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k=0; k<2*n-1; k++)
    S0: C[k] = 0;
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        S1: C[i+j] += A[i]*B[j];
```
Product of two polynomials:

- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k = 0; k < 2*n - 1; k++)
S0: C[k] = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
S1: C[i+j] += A[i]*B[j];
```
Introduction
Kernel Offloading
Polyhedral Model

Dependences

Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

\[
\begin{align*}
\text{for (int } k=0; k<2*n-1; k++) \\
S0: & \quad C[k] = 0; \\
\text{for (int } i=0; i<n; i++) \\
& \quad \text{for (int } j=0; j<n; j++) \\
S1: & \quad C[i+j] += A[i]*B[j];
\end{align*}
\]
Scheduling alternatives: loop reversal + interchange

Product of two polynomials:
- arguments in A and B;
- result in C.

```
for (int k = 0; k < 2*n - 1; k++)
S0: C[k] = 0;
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
S1: C[i+j] += A[i]*B[j];
```
Scheduling alternatives: loop reversal + interchange + tiling

Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k = 0; k < 2*n-1; k++)
S0: C[k] = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
S1: C[i+j] += A[i]*B[j];
```
Scheduling alternatives: loop skewing

Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for(int k=0; k<2*n-1; k++)
S0: C[k] = 0;
for(int i=0; i<n; i++)
  for(int j=0; j<n; j++)
S1: C[i+j] += A[i]*B[j];
```
Scheduling alternatives: loop skewing + tiling

Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```cpp
for (int k = 0; k < 2*n - 1; k++)
S0: C[k] = 0;
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
S1: C[i+j] += A[i] * B[j];
```
Scheduling alternatives: loop skewing + tiling

Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```
for (int k=0; k<2*n-1; k++)
S0: C[k] = 0;
for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
      S1: C[i+j] += A[i]*B[j];
```

+ possibility of intra-tile parallelism.
Mathematical Framework

Polyhedral Model
⇒ Parametric Integer Linear Programming
⇒ Presburger arithmetic

Means all of first order logic is computable\(^2\)!
- Set union, intersection, difference;
- Set projections;
- Equality, inclusion tests.

Even more:
- Transitive closure approximation
- Graph algorithm on infinite graphs!

\(^2\)might take a long long time... but rarely
Limitations...

Restrictions:

- Any sequence/nesting of if and for loops.
- Multidimensional arrays of direct data (no pointers).
- Static and affine control:
  - Boolean conditions for if: affine.
  - Loop bounds of for loops: affine.
  - Increments of for loops: constant value.
  - Array access functions: affine.

Exemples:

- Polyédriques: algèbre linéaire dense, Cholesky, stencils, ...
- Non polyédriques: FFT, algèbre linéaire creux, ...
Approximations and more

Approximations allows:

- non affine if conditions;
- non affine array access functions;
- parametric for loop increments.

We can deal with while loops and crazy spaghetti code with goto when they behaves in an approximately affine way. Abstract Interpretation

We can detect and use semantic properties like associativity and commutativity to do more transformations. Semantic Tiling
Conclusion

Developing a tool:
- Source-to-source compilation;
- Automatic parallelization and cache optimization;
- Automatic tiling (data locality);
- Automatic pipelining and communication coalescing;
- Automatic memory allocation (for local buffers);
- Based on ISL (Integer Set Library);
- Allowing approximations.

Developing better theory:
- Parametric tiling;
- Generalized lattice based memory folding;
- Scheduling/analysis of sequential/parallel languages.
Questions ?