CS217 Lecture 6

Case Study: Dot Product

Fall 2018

Plan of Today

- Overview of dot product
- A Software implementation of dot product
- A Hardware design of dot product
- Optimizing dot product
- How to understand the performance

What is Dot Product

- A cumulative sum of two equal-length sequences
  - $a = [a_1, a_2, a_3, \dots a_n]$
  - $b = [b_1, b_2, b_3, \dots, b_n]$
  - $DP_{a,b} = \sum_1^n a_i \cdot b_i$
- Geometric meaning
  - $\cos(a, b)$

Step 1: Software Simulation

```
"""
This little program simulates a dot product
between two randomly generated vectors
"""
import numpy as np

# Set up the data
size = 256
aIn = np.random.randint(0, size, size=size)
bIn = np.random.randint(0, size, size=size)

# Perform DP
accum = 0
for a, b in zip(aIn, bIn):
    accum += a * b

# Check if the results match
gold = np.dot(aIn, bIn)
cksum = gold == accum
print("PASS: ", cksum, "(DotProduct)")
```

$$a = [a_1, \dots, a_n]$$
$$b = [b_1, \dots, b_n]$$

$$DP_{a,b} = \sum_{1}^{n} a_i \cdot b_i$$

Step 1: Software Simulation

```
"""
This little program simulates a dot product
between two randomly generated vectors
"""
import numpy as np

# Set up the data
size = 256
aIn = np.random.randint(0, size, size=size)
bIn = np.random.randint(0, size, size=size)

# Perform DP
accum = 0
for a, b in zip(aIn, bIn):
    accum += a * b

# Check if the results match
gold = np.dot(aIn, bIn)
cksum = gold == accum
print("PASS: ", cksum, "(DotProduct)")
```

```
// In Spatial
// Set up the data
val size = 256
val aIn =
  Array.fill(size){ random[Int](size) }
val bIn =
  Array.fill(size){ random[Int](size) }

val gold =
  aIn.zip(bIn){_*_}.reduce{_+_}
```

Step 2: Design the HW

- What components do we need?
  - Let's say that we are working on integer arrays.

```
type T = Int
```

  - Memory Transfer
    - Off-chip Memory. What are the off-chip components?

Step 2: Design the HW

- What components do we need?
  - Let's say that we are working on integer arrays.

```
type T = Int
```

  - Memory Transfer
    - Off-chip Memory
      - Two DRAMs to store array a, b
      - One ArgOut register to store the dot product result

Step 2: Design the HW

- What components do we need?
  - Let's say that we are working on integer arrays.

```
type T = Int
```

  - Memory Transfer
    - Off-chip Memory

```
val a = DRAM[T](size)
val b = DRAM[T](size)
setMem(a, aIn)
setMem(b, bIn)
val result = ArgOut[T]
```

Step 2: Design the HW

- What components do we need?
  - Let's say that we are working on integer arrays.

```
type T = Int
```

  - Memory Transfer
    - Off-chip Memory
    - On-chip Memory
      - We need on-chip memory to load from DRAM
      - What is the memory component?

Step 2: Design the HW

- What components do we need?
  - Let's say that we are working on integer arrays.

```
type T = Int
```

  - Memory Transfer
    - Off-chip Memory
    - On-chip Memory
      - SRAM to load from DRAM

Step 2: Design the HW

- What components do we need?
  - Let's say that we are working on integer arrays.

```
type T = Int
```

  - Memory Transfer
    - Off-chip Memory
    - On-chip Memory

```
val aTile = SRAM[T](tileSize)
val bTile = SRAM[T](tileSize)
```

Step 2: Design the HW

- What components do we need?
  - Let's say that we are working on integer arrays.

```
type T = Int
```

  - Controller
    - What controllers do we need?
      - A controller that generates indices for loading from DRAM
      - A controller that generates indices for loading from SRAM

Step 2: Design the HW

- What components do we need?
  - Let's say that we are working on integer arrays.

```
type T = Int
```

  - Controller
    - What controllers do we need?
      - A controller that generates indices for loading from DRAM

```
val accum = Reg[T](0.to[T])
Sequential.Foreach (size by tileSize) { i =>
  val aTile = SRAM[T](tileSize)
  val bTile = SRAM[T](tileSize)
  Parallel {
    aTile load a(i::i+tileSize par innerPar)
    bTile load b(i::i+tileSize par innerPar)
  }
}
```

Step 2: Design the HW

- What components do we need?
  - Let's say that we are working on integer arrays.

```
type T = Int
```

  - Controller
    - What controllers do we need?
      - A controller that generates indices for loading from DRAM
      - A controller that generates indices for loading from SRAM

```
Sequential.Foreach (tileSize by 1) { ii =>
  accum := aTile(ii) * bTile(ii) + accum.value
}
```

Step 2: Design the HW

- What components do we need?
  - Let's say that we are working on integer arrays.

```
type T = Int
```

  - Register Interface
    - We need to send the content in accum back to the host. How can we do that?

Step 2: Design the HW

- What components do we need?
  - Let's say that we are working on integer arrays.

```
type T = Int
```

  - Register Interface
    - We need to send the content in accum back to the host. How can we do that?

```
val result = ArgOut[T]

result := accum.value
```

Step 2: Design the HW

- Overall Design

```
@spatial
object DotProductDemo extends SpatialApp {
  type T = Int

  def main() {
    val size = 256
    val aIn = Array.fill(size){
          random[T](size) }
    val bIn = Array.fill(size){
          random[T](size) }
    val a = DRAM[T](size)
    val b = DRAM[T](size)
    setMem(a, aIn)
    setMem(b, bIn)
    val result = ArgOut[T]
    Accel {
      val accum = Reg[T](0.to[T])
      Sequential.Foreach (size by tileSize) { i =>
        val aTile = SRAM[T](tileSize)
        val bTile = SRAM[T](tileSize)
        Parallel {
          aTile load a(i::i+tileSize)
          bTile load b(i::i+tileSize)
        }
        Sequential.Foreach (tileSize by 1) { ii =>
          accum := aTile(ii) * bTile(ii) + accum.value
        }
      }
      result := accum.value
    }
    val accelResult = getArg(result)
        val gold = aIn.zip(bIn){_*_}.reduce{_+_}
        val cksum = gold == accelResult
        println("PASS: " + cksum + "(DotProductDemo)")
    }
}
```

Step 2: Design the HW

- Overall Design

```
@spatial
object DotProductDemo extends SpatialApp {
  type T = Int

  def main() {
    val size = 256
    val aIn = Array.fill(size){
          random[T](size) }
    val bIn = Array.fill(size){
          random[T](size) }
    val a = DRAM[T](size)
    val b = DRAM[T](size)
    setMem(a, aIn)
    setMem(b, bIn)
    val result = ArgOut[T]
```

```
    Accel {
      val accum = Reg[T](0.to[T])
      Sequential.Foreach (size by tileSize) { i =>
        val aTile = SRAM[T](tileSize)
        val bTile = SRAM[T](tileSize)
        Parallel {
          aTile load a(i::i+tileSize)
          bTile load b(i::i+tileSize)
        }
        Sequential.Foreach (tileSize by 1) { ii =>
          accum := aTile(ii) * bTile(ii) + accum.value
        }
      }
      result := accum.value
    }
    val accelResult = getArg(result)
      val gold = aIn.zip(bIn){_*_}.reduce{_+_}
      val cksum = gold == accelResult
      println("PASS: " + cksum + "(DotProductDemo)")
    }
  }
```

Locality?

Step 2: Design the HW

- Question:
  - Do we have to use Sequential.Foreach controller?
    - Look at the way we calculate the product sum. Can we describe it using some other pattern?

Step 2: Design the HW

- Question:
  - Do we have to use Sequential.Foreach controller?
    - Look at the way we calculate the product sum. Can we describe it using some other pattern?
      - Dot product is essentially a reduction

Step 2: Design the HW

- Question:
  - Do we have to use Sequential.Foreach controller?
    - Look at the way we calculate the product sum. Can we describe it using some other pattern?
      - Performing dot product is essentially a reduction process
  - *Sequential* enforces the entire hardware to work on a tile / time
    - We want to overlap tile loading with the computation
    - Remove Sequential to enable pipelining

Step 2: Design the HW

- DotProduct with Reduce:

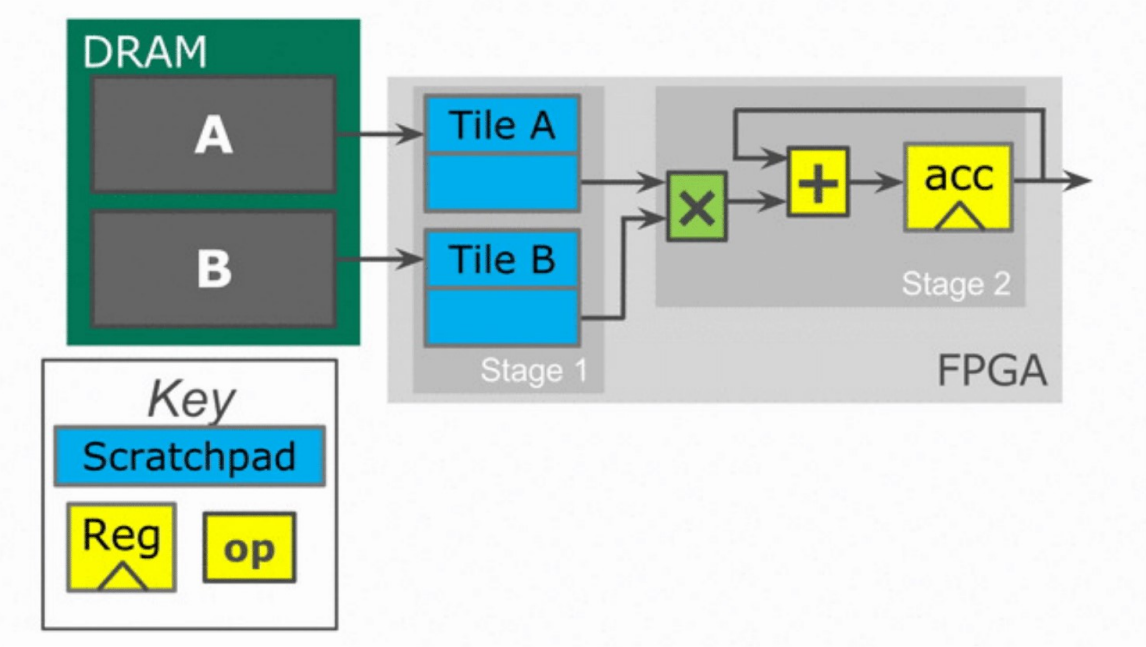
```
object DotProductDemo extends SpatialApp {
  type T = Int
  @virtualize
  def main() {
    val size = 256
    val aIn = Array.fill(size){
          random[T](size) }
    val bIn = Array.fill(size){
          random[T](size) }
    val a = DRAM[T](size)
    val b = DRAM[T](size)
    setMem(a, aIn)
    setMem(b, bIn)
    val result = ArgOut[T]
    Accel {
      val accum = Reg[T](0.to[T])
      Reduce(accum)(size by tileSize) { i =>
        val aTile = SRAM[T](tileSize)
        val bTile = SRAM[T](tileSize)
        Parallel {
          aTile load a(i::i+tileSize)
          bTile load b(i::i+tileSize)
        }

        Reduce(0)(tileSize by 1){ ii =>
          aTile(ii) * bTile(ii)
        }{_+_}
      }{_+_}

      result := accum.value
    }
    val accelResult = getArg(result)
    val gold = aIn.zip(bIn){_*_}.reduce{_+_}
    val cksum = gold == accelResult
    println("PASS: " + cksum +
          "(DotProductDemo)")
  }
}
```

Step 2: Design the HW

- Overall Design



Step 3: Optimization

- Parallelization
  - We can set parallelization factors for controllers, memory loads/stores
    - What happens when parallelizing controllers?
      - Extending the counter value to hold multiple consecutive values at once.
      - Physically banking the memory for reading all the desired value

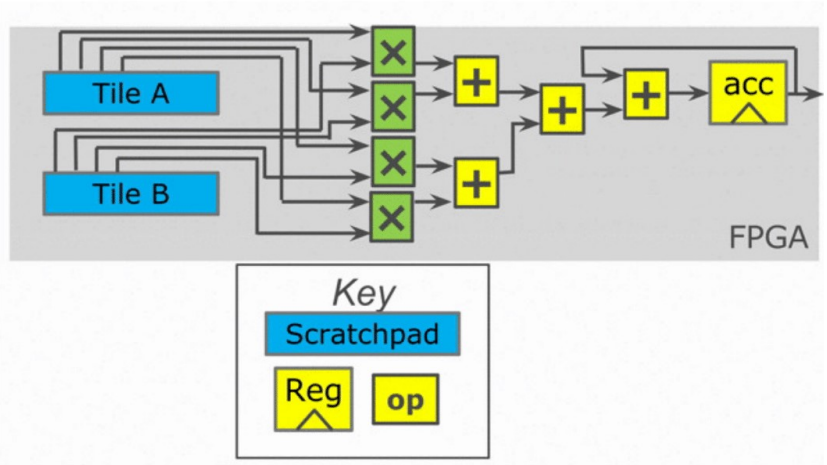
Step 3: Optimization

- Parallelization
  - Example: inner reduction in dot product, par by 4

```
val innerPar = 4
Reduce(0)(tileSize by 1 par innerPar){ ii =>
    aTile(ii) * bTile(ii)
}{_+_}
```

Step 3: Optimization

- Parallelization
  - Example: inner reduction in dot product, par by 4
    - Generated circuit



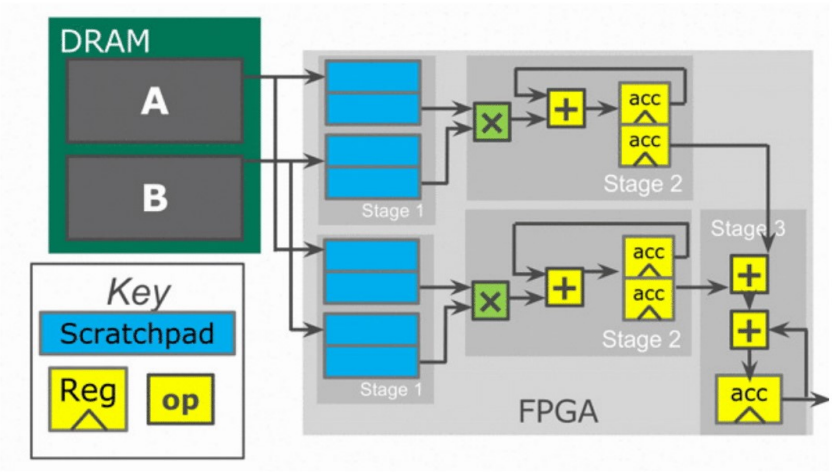
Step 3: Optimization

- Parallelization
  - Example: outer reduction in dot product, par by 2

```
val outerPar = 2
Reduce(accum)(size by tileSize par outerPar) { i =>
```

Step 3: Optimization

- Parallelization
  - Example: outer reduction in dot product, par by 2
    - Generated circuit



Step 3: Optimization

- Parallelization Example
  - Parallelize the outer Reduce controller by 2 (outerPar)
  - Parallelize the inner Reduce controller by 4 (innerPar)

```
Accel {
  val accum = Reg[T](0.to[T])
  Reduce(accum)(size by tileSize par outerPar) { i =>
    val aTile = SRAM[T](tileSize)
    val bTile = SRAM[T](tileSize)
    Parallel {
      aTile load a(i::i+tileSize)
      bTile load b(i::i+tileSize)
    }

    Reduce(0)(tileSize by 1 par innerPar){ ii =>
      aTile(ii) * bTile(ii)
    }{_+_}
  }{_+_}
  result := accum.value
}
```

Step 3: Optimization

- Parallelization
  - We can set parallelization factors for controllers, memory loads/stores
    - Upside: increases performance of your design
    - Downside: a parallelization of factor n essentially creates n copies of your design. May run out of resources

Step 4: Performance Analysis

- What do we care about?
  - Cycles needed to finish the design
  - Resource utilization

Step 4: Performance Analysis

- What do we care about the design?
  - Cycles needed to finish the design

	DP, SeqForeach	DP, Reduce,1,1	DP, Reduce,2,4
Cycles	7936	2340	1376

Step 4: Performance Analysis

- What do we care about?
  - Resources needed

	DP, SeqForeach	DP, Reduce,1,1	DP, Reduce,2,4
Cycles	7936	2340	1376
FixPt Ops	4	8	51
Block Mem (Bits)	2048	8192	16384
Reg (Bits)	32	128	224

Step 4: Performance Analysis

- What can we observe from this table?
  - Larger SRAMs improve data locality.
  - Adding parallelization factors improves the latency by ~2.

Step 4: Performance Analysis

- Analyzing Code

```
Accel {
  val accum = Reg[T](0.to[T])
  Reduce(accum)(size by tileSize par outerPar) { i =>
    val aTile = SRAM[T](tileSize)
    val bTile = SRAM[T](tileSize)
    Parallel {
      aTile load a(i::i+tileSize)
      bTile load b(i::i+tileSize)
    }

    Reduce(0)(tileSize by 1 par innerPar){ ii =>
      aTile(ii) * bTile(ii)
    }{_+_}
  }{_+_}
  result := accum.value
}
```

Step 4: Performance Analysis

- Analyzing Code

```
Accel {
  val accum = Reg[T](0.to[T])
  Reduce(accum)(size by tileSize par outerPar) { i =>
    val aTile = SRAM[T](tileSize)
    val bTile = SRAM[T](tileSize)
    Parallel {
      aTile load a(i::i+tileSize)
      bTile load b(i::i+tileSize)
    }

    Reduce(0)(tileSize by 1 par innerPar){ ii =>
      aTile(ii) * bTile(ii)
    }{_+_}
  }{_+_}
  result := accum.value
}
```

We can try to bring in all the data at once. Try tileSize = 256

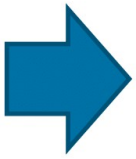
We want to parallelize the inner as much as possible. In this case. Try innerPar = 128

Step 4: Performance Analysis

- Analyzing Code

```
Accel {
  val accum = Reg[T](0.to[T])
  Reduce(accum)(size by tileSize par outerPar) { i =>
    val aTile = SRAM[T](tileSize)
    val bTile = SRAM[T](tileSize)
    Parallel {
      aTile load a(i::i+tileSize)
      bTile load b(i::i+tileSize)
    }

    Reduce(0)(tileSize by 1 par innerPar){ ii =>
      aTile(ii) * bTile(ii)
    }{_+_}
  }{_+_}
  result := accum.value
}
```



What kind of parallelism are we exploiting?

Step 4: Performance Analysis

- What can we conclude from this table?
  - Let's try 1 bigger design.
    - outerPar = 2
    - innerPar = 128
    - tileSize = 256

Step 4: Performance Analysis

	DP, SeqForeach	DP, Reduce,1,1	DP, Reduce,2,4	*Reduce, 2,128,256*
Cycles	7936	2340	1376	***1080***
FixPt Ops	4	8	51	***515***
Block Mem (Bits)	2048	8192	16384	***4.2M***
Reg (Bits)	32	128	224	***96***

Step 4: Performance Analysis

- Questions:
  - If we increase the tileSize, what resources would run out first?
  - If we increase the parallelization factor, what resources would run out first?
  - By looking at the resource utilization and the implementation, what could likely be the bottleneck?

Step 4: Performance Analysis

- Questions:
  - If we increase the tileSize, what resources would run out first?
    - BRAM
  - If we increase the parallelization factor, what resources would run out first?
    - FixPt Opts
  - By looking at the resource utilization and the implementation, what could likely be the bottleneck?
    - The outer Reduction. This design only needs 2 cycles to execute the inner reduction, but the outer reduction needs far more cycles to complete the accumulation.
    - Allocating more resource for the inner reduction won't improve the cycle number. The bottleneck is at the outer reduction.

Step 4: Performance Analysis

- How will you improve the dot product example?

Questions?