CyberGIS Education Modules

Release 1.1.1

M McKenney

August 18, 2016

A PDF version of the module: PDF

An Epub version of the module: EPUB

CHAPTER

ONE

INTRODUCTION TO THE MAPREDUCE MODEL OF PARALLELISM

1.1 Preliminaries

Creator Name: Mark McKenney

Content Title: Introduction to the MapReduce Model of Parallelism

Learning Objectives:

1. Describe the basic ideas of the mapReduce paradigm
2. Be able to construct mapReduce computations in scripting languages
3. Gain basic skills in: using the command line interface for MongoDB, loading a data set into MongoDB, performing basic queries to explore aspects of that data set.
4. Describe the basic architecture of a mapReduce system.
5. Describe the implications of adding/removing hardware to a mapReduce system.
6. Describe the data model of a Mongo/BigTable/Hive type system.
7. Be able to construct and execute a mapReduce porgram on MongoDB
8. Be able to perform queries to explore the result of a mapReduce query.

1.2 The mapReduce Paradigm

Section Goals:

1. Describe the basic ideas of the mapReduce paradigm
2. Be able to construct mapReduce computations in scripting languages

The original concept of mapReduce has its roots in functional programming. Basically, a **map** will apply a function to a *set* of input data, transform this data in some way, and **emit** the transformed data. A **reduce** will apply a function to a *set* of input data, and essentially combine that input data in some way so that it returns a smaller *set* of output data. Lets see an example:

Example

The following python program contains a map function that applies a function to each element of a list that will increment that element. The reduce function then sums the elements. The function being applied in the **map** is an anonymous function (i.e., a function that has no name and is not formally declared in an outer scope. So, the function is only defined for this line of code). The lambda function in the **map** takes one argument **x** and increments it, returning

the incremented value. Similarly, the lambda function in the **reduce** takes two arguments, **x** and **y**, and returns their sum.

```
>>> newNums = map( lambda x:x+1, nums )
>>> print newNums
[2, 3, 4, 5, 6]
>>> sum = reduce( lambda x,y: x+y, newNums )
>>> print sum
20
```

Notice that the lambda function in the **map** operates on a single item of the list of numbers. Therefore, each item in the list can be processed by the lambda function in parallel. This is referred to as an **embarrassingly parallel problem** because there are NO data dependencies among the list items and the application of the lambda function.

Similarly, the lambda function in the **reduce** can execute over the data items in the list in parallel, but not to the same degree as the **map**, since it requires two data elements. Both of the following execution plans are valid for the **reduce** above, but one will have more parallelism than the other:

$((((2+3)+4)+5)+6)$

$(2+3)+(4+5)+6$

Notice in the second plan that the $2+3$ and $4+5$ can execute in parallel.

The functional programming model of mapReduce sums up the behavior of a mapReduce computation very nicely, from the programmer's perspective. In fact, executing a massive mapReduce computation over thousands of compute nodes is essentially identical to executing the python program above, from the programmer's point of view; all that is required is a *mapper* and a *reducer*, and the system will take care of the rest.

1.3 Extending the mapReduce Model to Massive Parallelism

Section Goals:

1. Describe the basic architecture of a mapReduce system.
2. Discuss the implications of adding/removing hardware to a mapReduce system.
3. Describe the data model of a Mongo/BigTable/Hive type system.

1.3.1 Basic Big Data Concepts and Definitions

Big data is a general term applied to a large variety of situations, and with numerous definitions. In general, big data concepts tend to come into play when one or more of the **three V's** of big data begin to be expressed in a particular problem or application. The three V's of big data are:

- Volume: there is a lot of data. Typically things get interesting when the terabyte mark is exceeded, and get more interesting from there. Traditional databases and traditional data techniques can generally handle up to a TB of data without having to get too complex (in terms of guaranteeing uptime, data duplication and backups, and performance).
- Velocity: The data is generated, or arrives at the data center, very quickly. Anything that generates lots of data in a short time. For example, experiments in the Large Hadron Collider create up to PB of data in less than second. Facebook users are updating their accounts all the time.
- Variety: The data is not a fixed format, fixed size, or even consists of a fixed type of data. Traditional databases require the declaration of relations with explicit structure, and usually, fixed size records. This is impossible in many big data settings. Perhaps the data being stored evolves over time, perhaps the structure of records evolves

over time, perhaps not all records are complete. The system must be able to not only handle such data, but must be able to query data in a variety of formats

Big data systems wrestle with problems of scale that are just not an issue in smaller systems. Systems may comprise thousands of nodes. Even if the nodes are expensive, server grade hardware, node failures are going to be a problem. Think about creating a new social networking application that stores user data. The company requires a lot of storage, to store a lot of data items that must be quickly retrievable, that must provide some guarantee that data will not be lost, and that must do all this on a budget. Thus, central design goals are typically to be:

1. Scalable: be able to grow the systems with minimal configuration, and with heterogeneous hardware.
2. Maintainable in the face of failures: if a node fails, the data stored on that node must already be duplicated or triplicated on other nodes, and the system should notice when nodes fail and make sure all data that was stored on that node is sufficiently replicated on other nodes. This must be done automatically.
3. Use commodity, heterogeneous hardware: nodes need to be flexible, the system should be able to grow without upgrading existing nodes if desired.
4. Provide fast query and retrieval: the system needs some query mechanism that is highly parallel, flexible, and relatively easy to use. Furthermore the queries must be able to possibly return huge results (more data than can fit on a single node, possibly).

To achieve these goals, big data systems have traditionally been designed as a distributed file system combined with a query mechanism

1.3.2 Google File System (GFS)

[GFS2003]

GFS was what really got the whole big data thing going on a large commercial scale. The idea was to create a file system that can expand over multiple nodes (thousands) automatically, can automatically incorporate new nodes and identify and de-incorporate faulty nodes, and can handle huge volumes of data.

Remember the Google application area. Web pages are crawled and stored on Google servers. The term **crawled** means that a copy of the web page is downloaded to the Google servers, and all the links on that page are stored so that they can be downloaded as well. The downloaded web pages are then processed into a form that is easily queryable using massive parallelism. Crawlers are only so fast (see exercise below), and web pages need only be re-crawled so often (see exercise below), so the velocity of data coming into the servers is not necessarily huge. But, they are keeping a copy of the Internet, so volume is huge! Also, web pages are unstructured text, contain multimedia, etc., so variability is huge.

Exercise

Web crawlers are only so fast. What limits the speed of crawling the web? Think of the architecture of the Internet, the destination of all of those web pages that are crawled, bottlenecks, etc.

Web pages do not necessarily need to be re-crawled very often. What would be an exception to this rule? What are the consequences of re-crawling more often vs. less often.

1.3.3 Basic Architecture of the GFS (and other similar distributed file systems)

The architecture of GFS and other similar file systems (Hadoop), is best understood if considered in the context of the assumptions under which the systems are designed. These include:

1. The system is built from inexpensive, failure-prone hardware.
2. The system must be built from heterogeneous hardware. (the compute nodes do not need to be identical)

3. The system must be easy to maintain. In other words, adding nodes or removing nodes should be easy. In fact, adding hundreds or thousands of nodes should require minimal software configuration.
4. High bandwidth is more important than low latency. There are very few, stringent response time requirements.

So, the distributed file system should be easy to manage on the hardware side, and the mapReduce interface should make it easy to run massively parallel programs from the software side. In essence, everything should be easy. Think about this. A huge installation of such a file system may have thousands of inexpensive, failure-prone nodes; therefore, at any given time, at least one node may be broken. If adding a node to, or removing a node from, a system required significant hardware and software configuration, the costs of paying people to do this would become prohibitive at large scales. Also, nodes may not be able to be replaced before another node breaks, meaning that the system is always running below peak capacity.

To achieve these goals, the architecture is remarkably simple. In its most basic form, such a distributed file system contains 1 **master node**, and many **file server nodes**. The master node basically keeps track of which file servers contain the files stored in the system. When a piece of information (lets say a copy of a webpage) needs to be inserted into the system, the master node checks to see which file servers have space for it. The master node then sends the file containing the web page to n file servers where n is the desired amount of duplication. The file servers simple store the file.

The assumptions 1,2, and 3 are all satisfied by this basic architecture. Note that no hardware or software requirements are imposed on the master or file servers. The only requirement is that the GFS code be installed. There is no requirement for servers to be identical, or have any special features. Finally, data is stored on n different physical servers. Lets assume $n = 3$, all data is stored in triplicate. If one server containing a particular piece of data breaks and stops responding, the master will notice this, and realize it only has two copies of all data that was stored on that server. The master will instruct the file servers that contain other copies of that data to copy that data to another file server so that data is then, again, stored in triplicate. If a new node is added to the system, the master must simply be told of its existence, and it automatically begins to use it. Clearly, such a file system enables relatively easy management of a huge number of servers.

There is one more piece to the architecture, and that is the **client**. Essentially, a **client** is written to execute a particular type of query. For example, lets say that a common query will be to count the number of web pages that contain the word *cat*. However, we want to generalize this so that we can count the number of web pages that contain any user specified word. A programmer can write a client that will do the following:

1. Get a query word from the user. Lets call this x.
2. The client will go to the master node to find out which web pages are stored in the system and which file server each web page is on. The client will then go to each file server it needs to visit, read the web page, and count it if it contains the word x provided by the user.
3. The client will sum the results into a final count.

The important point to notice is that once a client is created, any user can simply use the client. This makes querying the system super easy if the appropriate client exists. Most file systems of this type contain a mapReduce client.

Finally, notice that since the data is on independent servers, all servers can be accessed in parallel. In the example where we count the number of web pages containing the word *cat*, each server can compute a count independently, and the counts for various servers can then be merged (If you look back at the Python map reduce example, you should be able to see how a mapper can be applied to each server independently, and results from each server get summed in a reducer!)

1.3.4 Using a mapReduce Client

To execute a mapReduce query, we assume that we have a working Hadoop/GFS/other system set up that provides a mapReduce client. Each system will have its own specifics over how we need to write our query, but in general, they all follow the same form:

1. We need a map function that will be executed on every file in the system. That function will **emit** a **key,value** pair to the client. The client will automatically group values together with matching keys.
2. We need a reduce function that will accept a key, and an array of values that were emitted with that key from the mapper. The reducer will then perform some computation, and return another **key,value** pair. Again, the client will store those in the system.
3. The result is a file containing a bunch of **key,value** pairs.

Example

Map and reduce functions have the following signatures:

$map(k1, v1) \rightarrow list(k2, v2)$

$reduce(k2, list(v2)) \rightarrow list(v3)$

So, constructing mapReduce queries is easy for the programmer, just make 2 functions and don't worry about parallelism or synchronization. The client takes care of that for you.

Lets look again at an example similar to the cat counting example. In this example, lets assume we want to find the number of times ALL words occur in our data set:

Example

Assume we have a working distributed file system that provides a mapReduce client. Further, we are storing files in the file system such that each file contains the contents of a web page, and the file name of each file is the URL of the web page. If we want to write a mapReduce query to count the number of times all words appear in those web pages, we could write the following query:

(note that this is a common example query used in *[GFS2003]* and other places)

```
map(String key, String value):
// key: file name
// value: file contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
// key: a word
// values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

1.4 Getting MongoDB Up and Running

In this module, we use MongoDB as an example for a few reasons:

1. It is open source and freely available.
2. It is available on a variety of platforms.
3. A basic, yet usable, installation requires almost no configuration.
4. It can scale easily across multiple file servers, or run on a single machine.

5. The mapReduce functions are written in javascript, a language that is somewhat accessible to novice programmers.
6. MongoDB is lightweight enough that students can install and use an instance on their own laptops/desktops.

Here, we will briefly list the steps to install MongoDB on a working Linux server. The MongoDB website has excellent documentation for installing a basic MongoDB instance on a variety of platforms. If you are an instructor, your Instructional Technologies Support office should be able to provide you with a basic Linux server and user accounts for your students. This tutorial uses the Ubuntu package manager, you will need to adapt to your own Linux distribution.

Once you have a Linux server, you need to install the MongoDB system:

```
user@server/~$ sudo apt-get install mongodb
[sudo] password for user: [enter your password]
```

The above commands will install a basic 1 node MongoDB on your computer. At this point, you may need to start the MongoDB, but most package managers will take care of this for you. If there is trouble, check the MongoDB website tutorials for help.

At this point the MongoDB instance is avaialable to ALL users on the system. To enter the command line interface to the database, you simply type `mongo`. The MongoDB instance can contain multiple **databases**. Each database will have a name, and will contain **collections** of **documents**. To see the databases currently housed in your MongoDB instance, try the following:

```
user@server/~$ mongo
MongoDB shell version: 2.4.9
connecting to: test
> show dbs
local 0.078125GB
> quit()
```

The `mongo` command starts the MongoDB shell, which you can use to execute commands on the database. MongoDB automatically connects the `test` database (if that database does not exist, it will be created once you create a collection and documents in it). The `show dbs` command shows all the databases currently housed in the MongoDB instance. The local database is created by MongoDB and Mongo uses it for bookkeeping purposes. Each line beginning with a > is a command prompt. The `quit()` command allows you to exit the MongoDB instance

---

Security Note!!

By default, ALL users that have access to the Linux server effectively have administrator privileges on the MongoDB instance. There are ways to limit this, but for the purposes of class assignments, it is not really necessary. However, you need to emphasize to students that they should create databases with names that are unique (preface the name with your student ID number, for example) so they do not accidentally clobber each other's databases. Also, students should be instructed not to delete other student's databases!!

1.5 The MongoDB Data Model

As we mentioned earlier, a MongoDB instance contains many **databases**. One database contains many **collections**. A collection contains many **documents**. The document is the primary mechanism of structuring data in MongoDB.

Mongo documents are simply JSON types. JSON is a text based data representation that allows a user to define structure in data, but does not impose structure on data. It is basically a list of labels and values:

Example of a JSON document

Lets say I want to create a document that describes a grocery store. In fact, I am trying to keep a database of grocery stores, so each document will contain information about 1 grocery store. A simple example might be:

```
{
  "name" : "Superman's Supermarket:,
  "address" : "123 Super St.",
  "NumberOfEmployees" : 20
}
```

Now, lets say that I want to add something a bit more complex. We need to keep track of a store's specialty items, but a store may have more than 1 specialty item, or none at all. The nice thing about a JSON document is that a JSON document can be a value in a JSON document. Lets create a competitor's store:

```
{
  "name" : "Batman's Supermarket:,
  "address" : "123 Bat Cave Rd",
  "NumberOfEmployees" : 15,
  "SpecialItems" : {
    "Fruit" : "blueberries",
    "PetFood" : "Bat Chow"
  }
}
```

The **specialItems** label contains an entire sub-document. In general, any document can contain a subdocument, which can contain a subdocument, etc. See http://json.org/ for all the details of JSON.

One thing to note is that a Mongo collection is defined as containing documents, meaning, any document is fine. In the example above, both documents can be stored in the same collection, despite having different structure, different labels, different size, etc. In fact, Mongo enforces NO constraints on the documents by default. If you create 100 documents each representing a supermarket, they are not required to share even a single label!

Concept

Because no constraints on the type, structure, or size of documents in a collection are enforced by default, it is up to the user to know the structure of the documents they are querying (or least have the ability to find out the structure).

1.6 Loading a Data Set Into MongoDB

Section Goals

1. Gain basic skills in: loading a data set into MongoDB.

At this point, any user with access to the server on which the MongoDB instance is installed can log into the server, and then manipulate create/delete/query data on the MongoDB instance. Lets assume a student, Alice, is going to load some data for an assignment. For this tutorial, we have provided a data file consisting of on-time-statistics for airlines in the United States. This data set is freely available from here: http://www.transtats.bts.gov/OT_Delay/OT_DelayCause1.asp A copy of the data containing entries ranging from 1987 -2014 is provided: `airlineDelay.csv` (30MB file!). We have made modifications to the raw data in 3 places: some of the entries in the raw data had blank values that were confusing the CSV file reader, so we modified those 3 lines so that the CSV file reader could read the data to load it into MongoDB. No data was added or removed, we simply changed the number of commas on those 3 lines.

To load the data set, you first need to get the `airlineDelay.csv` file onto the server where MongoDB is installed. It may be useful for the instructor to put the file in a world-readable directory on the server.

To create a database, enter the mongo shell and type: `use [your DB name]`. Again. . . all users have root access to the mongo database (although not to the sever). This means you have the ability to alter/destroy any data in the mongo system, including the data belonging to other students. Don't be mean, don't mess with other people's data, don't waste anybody's time. Everyone should create their own database and call it something unique; for example, your user name. Within your own database, you can create as many collections as you like.

To load the flight data into your database, use the following command from the command line (NOT the mongo shell!!) (replace [your DB name] with the name of your database within mongo. It will add the data to the collection called delayData):

```
user@server/~$ cd example
user@server/~/example$ ls
airlineDelay.csv
user@server/~/example$
user@server/~/example$ mongoimport -d userDB -c delayData --type csv --file airlineDelay.csv --header
connected to: 127.0.0.1
Thu Oct 16 10:07:43.007     Progress: 7483974/30791275  24%
Thu Oct 16 10:07:43.014       42000 14000/second
Thu Oct 16 10:07:46.010     Progress: 14274866/30791275 46%
Thu Oct 16 10:07:46.010       80200 13366/second
Thu Oct 16 10:07:49.011     Progress: 21239882/30791275 68%
Thu Oct 16 10:07:49.012       119600  13288/second
Thu Oct 16 10:07:52.007     Progress: 26615020/30791275 86%
Thu Oct 16 10:07:52.007       150000  12500/second
Thu Oct 16 10:07:53.822 check 9 173658
Thu Oct 16 10:07:54.664 imported 173657 objects
user@server/~/example$ mongo
MongoDB shell version: 2.4.9
connecting to: test
> show dbs
local 0.078125GB
userDB  0.453125GB
> quit()
```

Now, you have a database called `userDB` containing 173,657 documents taking up 0.078GB of space. This is not exactly huge data, but its big enough to practice with. Also, this is a real data set that you can use to learn about the real world!

Warning: Don't Be Mean!!
Remember, every user should create a unique name for their database, and NOT interfere with anyone else's database. One positive note is that if the MongoDB instance gets mangled, its very easy to delete the whole thing, re-install, and reload the data as we just did.

1.7 Basic MongoDB Queries to Explore Your Data Set

Section Goals

1. Gain basic skills in: using the command line interface for MongoDB and performing basic queries to explore aspects of that data set.

Each entry in the airline delay database represents the aggregate monthly data for a particular airline at a particular airport. For example, one entry might be the delay information for Southwest Airlines at St Louis Airport during December 2013.

Each entry lists the number of flights, the number of flights delayed (15 minutes beyond the scheduled arrival time), the number of flights canceled and diverted, the minutes of delay due to carrier delay - weather delay - national air

system delays - security delay. The _ct fields list the count of flights experiencing each delay (they add up the arr_del15 column). Because a flight can have multiple delay types, these numbers are real numbers. For example, if a flight was 30 minutes late, 15 minutes due to weather and 15 minutes due to security it will contribute .5 to weather_ct and .5 to security_ct. It will also contribute 15 to the weather_delay field and 15 to the security_delay field.

Finally, the arr_delay column is the sum of total delay minutes for the document (that is . . . arr_delay = carrier_delay+ weather_delay + nas_delay + security_delay + late_aircraft_delay)

The names of each field in an entry:

Label/field name	Meaning
year	The year the month occurred
month	The month for which data was collected
carrier	The airline code
carrier_name	The airline name
airport	The airport code
airport_name	The airport name
arr_flights	# of flights that arrived at the airport
arr_del15	# of flights that arrived >= 15 minutes late
carrier_ct	# of flights delayed due to the carrier
weather_ct	# of flights delayed due to weather
nas_ct	# of flights delayed due to national air system
security_ct	# of flights delayed due to security
late_aircraft_ct	# flights delayed because a previous flight using the same aircraft was late
arr_cancelled	# of canceled arrivals
arr_diverted	# of scheduled arrivals that were diverted
arr_delay	Sum of the delay minutes
carrier_delay	Total minutes of delays due to carriers
weather_delay	Total minutes of delays due to weather
nas_delay	Total minutes of delays due to natl. air service
security_delay	Total minutes of delays due to security
late_aircraft_delay	Similar to late_aircraft_ct. The total minutes of delay due to a previous flight using the same aircraft arriving late.

The JSON document structure for an individual document is shown by looking at a single imported document (1 row from the database). The following shell sequence shows how to use a command to see 1 document in a collection. Always remember to first switch to the database you want to use, then explore the collections. In this example, the database is named `airline` and the collection is `delays`:

```
user@server/~/example$ mongo
MongoDB shell version: 2.4.9
connecting to: test
> use airline
switched to db airline
> show collections
delays
system.indexes
> db.delays.findOne()
{
        "_id" : ObjectId("526ea45fe49ef3624c73e94f"),
        "year" : 2003,
        " month" : 6,
        "carrier" : "AA",
        "carrier_name" : "American Airlines Inc.",
        "airport" : "ABQ",
        "airport_name" : "Albuquerque, NM: Albuquerque International Sunport",
```

```
        "arr_flights" : 307,
        "arr_del15" : 56,
        "carrier_ct" : 14.68,
        "weather_ct" : 10.79,
        "nas_ct" : 19.09,
        "security_ct" : 1.48,
        "late_aircraft_ct" : 9.96,
        "arr_cancelled" : 1,
        "arr_diverted" : 1,
        "arr_delay" : 2530,
        "carrier_delay" : 510,
        "weather_delay" : 621,
        "nas_delay" : 676,
        "security_delay" : 25,
        "late_aircraft_delay" : 698,
        "" : ""
}
>
```

View documents in a collection

Assume a collection named `delays` in a selected database.

To view a single document in a collection, use:

```
> db.delays.findOne()
```

To view multiple documents in a collection, use:

```
> db.delays.find()
```

1.8 Constructing MapReduce Queries on Your Data Set

Section Goals

1. Be able to construct and execute a mapReduce program on MongoDB
2. Be able to perform queries to explore the result of a mapReduce query.

MongoDB provides a mapReduce client to execute queries across a database. If we only have a 1 node instance, then clearly there will be no parallelism across machines; however, recall that the nice thing about the mapReduce model and its associated distributed file systems is that we can easily add more machines to achieve more parallelism, and not have to change our queries at all to take advantage of it!

Mongo uses javascript to represent mapReduce queries. This is nice because javascript is generally easy to use, and provides just about everything you need as built in functions/libraries/types. It is also rather readable. If you have never used javascript, you will see that in the following examples that it will be somewhat intuitive for our uses.

The general prototype of a mapReduce operation in Mongo is the define a mapper function and reducer function, then use the mapReduce client (implemented as a Mongo function) to execute the query using the specified functions. For example:

Mongo mapReduce Prototype

The following code creates a mapper function named `mapper1` and a reducer function named `reducer1`. The result of the mapReduce operation will be a mongo **collection** named `OUTPUT_COLLECTION_NAME`.

```
var mapper1 = function() {
  YOUR CODE HERE
  emit( "SOMEKEY", SOMEVALUE );
};
var reducer1 = function( keyval, areasArray ) {
  YOUR CODE HERE
  return ( SOMEVALUE );
};
db.test.mapReduce( mapper1, reducer1,{out:"OUTPUT_COLLECTION_NAME"} )
db.OUTPUT_COLLECTION_NAME.find()
```

Note that the reducer does not explicitly return a key value. The key value passed to the reducer will automatically be associated with the value returned from the reducer by the mapReduce client.

Lets take a look at an actual query over the airline statistics data set. We will begin with a basic query that finds the airport in the united states that has had the most number of diversions throughout the entire data set. For each document in our collection, we will be concerned with 2 pieces of data:

1. The name of the airport: `airport_name`
2. The number of flights diverted from that airport: `arr_diverted`

Remember, that each document contains the data for a particular airline at a particular airport during a particular month. So, for the St. Louis Airport during January 2014, there will be a document containing statistics for Southwest Airlines, another document for Delta Airlines, another document for United Airlines, etc.

In order to construct the result for our query, we basically need to construct a collection containing a single JSON document for every airport in our data set. The document needs to contain the name of the airport, and the number of flights diverted from that airport. So, we need to transform the set of documents in our database into the desired set of documents containing the answer. All the data is in the database, it is just not is in a format in which the answer is readily available. We will use a mapReduce job to make this transformation.

Example: Find the airport with most number of diversions.

1. A document contains data for 1 airline at 1 airport during 1 month. We need to extract the data we need from each document. The mapper will do this:

```
var m1 = function(){
 emit( this.airport_name, this.arr_diverted );
};
```

NOTE the `this` keyword. A mapper is applied to every document in the data set. The `this` keyword allows us to access data from the document. `this.airport_name` retrieves the airport name from the document.

Recall that when we `emit` a value, the mapReduce client will automatically group all values with a matching key together into a list. So, the end of the mapper stage will result in a list of keys; each key will be associated with a list of values that were emitted with that key. For example, if there were 3 documents in the database indicating that STL had 3 diversions in January, STL had 2 diversion in February, and HOU had 7 diversion in January. In short, assume the following JSON documents in the delayData collection:

```
{
   "airport_name" : "STL",
   "year" : "2012",
   "Month" : "January",
   "arr_diverted" : "3"
 }

 { "airport_name" : "STL",
```

```
   "year" : 2012,
   "Month" : "February",
   "arr_diverted" : 2
 }

 { "airport_name" : "HOU",
   "year" : 2012,
   "Month" : "January",
   "arr_diverted" : 7
 }
```

Then, at the end of the emit stage the mapReduce client would construct the following lists:

Key	Value List
STL	[3,2]
HOU	[7]

The job of the reducer is now to reduce those lists into a meaningful result. Conceptually, a reducer will be run for each key. In practice, this is not necessarily true, as the system can take advantage of parallelism to schedule a more efficient execution plan (See the reducer example in Python up at the beginning). So, in concept, a reducer simply reduces all the values for a particular key into a single, meaningful, value. Remember, we want the total number of diversions for our airports.

The following reducer will sum up all the values in the list associated with a key. It will then return the sum. Notice the function `Array.sum()`; this is provided by javascript. You can always explicitly write a loop to compute the sum.

NOTE: the reducer does not explicitly emit a key,value pair. In Mongo, whatever value is in the `return` statement will be associated with the `key` passed into the function.

```
var r1 = function( key, valArr ) {
  return Array.sum( valArr );
}
```

Once the reducer is run, we should end up with the following keys and values:

Key	Value List
STL	[5]
HOU	[7]

Finally, we have defined the map and reduce functions. Now we must actually execute them. To begin a mapReduce client with the above mapper and reducer, we use the following code.

```
db.delayData.mapReduce( m1, r1, {out:'tmp'});
```

The `{out:'tmp'}` simply means that the collection containing the result of our query will be named `tmp`.

The last step is to look at the result. Remember, we wanted to find the airport with the most diversions, but our query just returned the number of diversions from ALL airports. The easiest way to find our answer at this point is to just sort the results. Mongo uses the `find()` function to return documents in a collection. Mongo also provides a `sort()` function to sort the documents in a collection. The following lien will sort the documents in the `tmp` collection in descending order and print them to the screen.

```
db.tmp.find().sort( {value:-1} )
```

All together, you can copy and paste the following code into the Mongo shell to execute the operations and view the results. Remember, this code assumes the data is in a collection called `delayData`. Replace `delayData` with whatever collection your data is in:

```
var m1 = function(){
  emit( this.airport_name, this.arr_diverted );
};

var r1 = function( key, valArr ) {
  return Array.sum( valArr );
}

db.delayData.mapReduce( m1, r1, {out:'tmp'});
db.tmp.find().sort( {value:-1} )
```

A script of the execution is shown below. Note that because we don't want to make anyone angry at our tutorial, we will show the airports with the LEAST number of diversions. Generally, this means the airports are either:

1. Not receiving many commercial flights.
2. Small (and don't get a lot of flights).
3. So remote that aircraft cannot divert because there is no place to divert to (the scarier option).
4. Do not report data for some reason.

```
user@server/~/example$ mongo
MongoDB shell version: 2.4.9
connecting to: test
> show dbs
local 0.078125GB
userDB  0.453125GB
> use userDB
switched to db userDB
> show collections
delayData
system.indexes
> var m1 = function(){
...   emit( this.airport_name, this.arr_diverted );
... };
>
> var r1 = function( key, valArr ) {
...   return Array.sum( valArr );
... }
>
> db.delayData.mapReduce( m1, r1, {out:'tmp'});
{
  "result" : "tmp",
  "timeMillis" : 4565,
  "counts" : {
    "input" : 173657,
    "emit" : 173657,
    "reduce" : 16838,
    "output" : 363
   },
  "ok" : 1,
}
> show collections
delayData
system.indexes
tmp
> db.tmp.find().sort( {value:1} )
{ "_id" : "Clarksburg/Fairmont, WV: North Central West Virginia", "value" : 0 }
{ "_id" : "Columbus, MS: Columbus AFB", "value" : 0 }
```

```
{ "_id" : "Dickinson, ND: Dickinson - Theodore Roosevelt Regional", "value" : 0 }
{ "_id" : "Greenville, MS: Mid Delta Regional", "value" : 0 }
{ "_id" : "Guam, TT: Guam International", "value" : 0 }
{ "_id" : "Gustavus, AK: Gustavus Airport", "value" : 0 }
{ "_id" : "Iron Mountain/Kingsfd, MI: Ford", "value" : 0 }
{ "_id" : "Kansas City, MO: Charles B. Wheeler Downtown", "value" : 0 }
{ "_id" : "Moses Lake, WA: Grant County International", "value" : 0 }
{ "_id" : "Pago Pago, TT: Pago Pago International", "value" : 0 }
{ "_id" : "Palmdale, CA: Palmdale USAF Plant 42", "value" : 0 }
{ "_id" : "Phoenix, AZ: Phoenix - Mesa Gateway", "value" : 0 }
{ "_id" : "Saipan, TT: Francisco C. Ada Saipan International", "value" : 0 }
{ "_id" : "Salem, OR: McNary Field", "value" : 0 }
{ "_id" : "Staunton, VA: Shenandoah Valley Regional", "value" : 0 }
{ "_id" : "Visalia, CA: Visalia Municipal", "value" : 0 }
{ "_id" : "Yakima, WA: Yakima Air Terminal/McAllister Field", "value" : 0 }
{ "_id" : "Del Rio, TX: Del Rio International", "value" : 1 }
{ "_id" : "Houston, TX: Ellington", "value" : 1 }
{ "_id" : "North Platte, NE: North Platte Regional Airport Lee Bird Field", "value" : 1 }
Type "it" for more
>
```

That was a long example, but it hits all the high points.

The next example emphasizes the fact that not documents must emit a value. For example, if we only want to find the number of diversions occurring at airports in 2012, we must only emit values from documents indicating that they have data for 2012. The following code changes the previous example slightly to accomplish this:

```
var m1 = function(){
  if( this.year == 2012) {
    emit( this.airport_name, this.arr_diverted );
  }
};

var r1 = function( key, valArr ) {
  return Array.sum( valArr );
}

db.delayData.mapReduce( m1, r1, {out:'tmp'});
db.tmp.find().sort( {value:-1} )
```

Finally, part of the power of mapReduce is that the key does not have to be a basic data type. In the Mongo system, this means that they key can be a JSON document. In generel, in other systems implementing a mapReduce framework, this means the key can be almost any complex grouping of data items.

For example, if we want to find the number of flight diversions that occur in St Louis, but we want to group them according to year, then we must construct our key to include the airport name and the year. Furthermore, we only emit such a key if the airport name is `STL`. The following example does this.

Example: Document as a Key:

Assume the following JSON documents in the delayData collection:

```
{ "airport_name" : "STL",
  "year" : 2012,
  "arr_diverted": 3
}

{ "airport_name" : "STL",
```

```
  "year" : 2012,
  "arr_diverted": 5
}

{ "airport_name" : "STL",
  "year" : 2011,
  "arr_diverted": 7
}

{ "airport_name" : "HOU",
  "year" : 2012,
  "arr_diverted": 7
}
```

The mapper must construct a key for each input document consisting of a JSON document that contains the airport name and year:

```
var m1 = function(){
  var rval = { airport: this.airport_name,
               year: this.year };
  if( this.airport=="STL") {
    emit( rval, this.arr_diverted );
  }
};

var r1 = function( key, valArr ) {
  return Array.sum( valArr );
}

db.delayData.mapReduce( m1, r1, {out:'tmp'});
db.tmp.find().sort( {'_id.year':-1} )
```

The mapper will return process each document, emitting keys and values. The mapReduce client will then construct the following lists associated with keys:

Key	Value list
{“airport”:”STL”, “year”:2012}	[3,5]
{“airport”:”STL”, “year”:2011}	[7]

Finally, the reducer will sum the lists it is given and return the results to the `tmp` collection.

1.9 Exercises

1. Query1: Find the total amount of delay minutes in the database (1 number).
2. Query2: Find the total amount of delay minutes grouped by airline.
3. Query3: Find the airport with the most canceled flights (this may require 2 map reduces, one to group the airports by their canceled flights, and another to find the airport with the most).
4. Query 4: Find the airport with the most canceled flights in 2012.
5. Query 5: Find, for each airline, the airport at which they have the most number of delays (use the arr_del15 column).
6. Query 6: Find the average delay time for the airport with the most flights in the database.

7. Query 7: For each airline, find the delay category that makes up the smallest amount of time of their delays, and then find the airport for which they have the most minutes of that delay category.

Hint for Query 7: One way to answer this will require 3 separate sets of map/reduce functions, 1 of those sets will get called in a loop:

step 1. get the correct category of delay for each airline

step 2. get the airline, airport, and sum of delays of the correct category for each airline/airport combo. This is where you probably need to execute map/reduces in a loop, once for each airline. Check out the 'scope' parameter to the db.mapReduce() function

step 3. find the airline/airport combo with the max number of delay minutes

1.10 Quick MongoDB Command Cheat Sheet

projection: (the second argument) >db.delays.find({},{weather_delay:1})

count docs in a collection: >db.runCommand({count:'delays'})

create database: > use mydb

drop database > use mydb; > db.dropDatabase();

list databases: > show dbs

drop collection: > db.collection.drop()

list a documents: > db.plots.findOne()

list all documents: > db.plots.find()

find price equal: > db.plots.findOne({price: 729})

find price greater than > db.plots.findOne({price: {$gt:729, $lt: 800}})

CHAPTER

TWO

REVIEW OF MULTITHREADED PROGRAMMING

2.1 Sum3 Problem Overview

2.1.1 Basic Description

The *Sum3* problem is described by a rather simple question: Given a set of n integers, how many triples of distinct elements in that list add up to 0.

For example, given the following list:

```
-1, -2, 0, 2, 3
```

the answer is 2:

```
-1 + -2 + 3 = 0
-2 + 0 + 2 = 0
```

The simplest way to code a solution to the *Sum3* problem is to use a double nested looping structure to generate the indices of all possible triples in the input array. Clearly, this is simple to express in code but has an unfortunate time complexity of $O(n^3)$. For example, the following C++ code uses 3 `for` loops to achieve a correct solution.

```
#include <iostream>
using namespace std;

int main( )
{
        int dataSize = 5;
        int* data = new int[ dataSize ];
        data[0] = -1;
        data[1] = -2;
        data[2] = 0;
        data[3] = 2;
        data[4] = 3;
        // do the naive Sum3 computation.  O(n^3)
        int count = 0;
        for (int i = 0; i < dataSize-2; i++)
                for (int j = i+1; j < dataSize-1; j++)
                        for (int k = j+1; k < dataSize; k++)
                                if (data[i] + data[j] + data[k] == 0)
                                        count++;

        cout<< count <<endl;

}
```

2.1.2 Complete Naive Solution

The next code block uses the same solution as above, but includes a command line parameter that will generate an array of random numbers of a user-specified size. The `getRandInt()` function loads the vector passed into it with random integers. The `data` vector allocates space for the number of integers required, and is then passed to the `getRandInt()` function. This version of the program allows the user to get a feel for how an $O(n^3)$ program behaves in practice. Try compiling the program and running it.

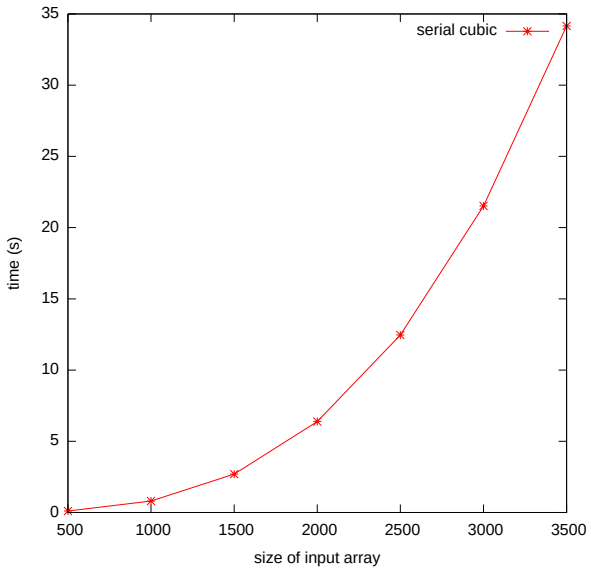
```
#include <iostream>
#include <sstream>

using namespace std;

int getRandInt( int* dataVec, int dataSize )
{
        // load up a vector with random integers
        int num;
        for( int i = 0; i < dataSize; i++ ) {
                // integers will be 1-100
                num = rand() % 100 +1;
                if( rand( ) % 2 == 0 ) {
                        // make some integers negative
                        num *= -1;
                }
                dataVec[i] = num;
        }
}


int main(int argc, char * argv[] )
{
        int dataSize = 0;

        if( argc < 2 ) {
                std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
                exit( -1 );
        }
        {
                std::stringstream ss1;
                ss1 << argv[1];
                ss1 >> dataSize;
        }
        if( argc >= 3 ) {
                std::stringstream ss1;
                int seed;
                ss1 << argv[2];
                ss1 >> seed;
                srand( seed );
        }
        else {
                srand( 0 );
        }

        // create a data vector
        int* data = new int[ dataSize ];

        // load it up with random data
        getRandInt( data, dataSize );
```

```
51          int *dataPtr = &data[0];
52          // do the naive Sum3 computation.  O(n^3)
53          int count = 0;
54          for (int i = 0; i < dataSize-2; i++)
55                  for (int j = i+1; j < dataSize-1; j++)
56                          for (int k = j+1; k < dataSize; k++)
57                                  if (dataPtr[i] + dataPtr[j] + dataPtr[k] == 0){
58                                          count++;
59                                  }
60          cout<< count <<endl;
61  }
```

Here are some running times for the previous program:

Array Size	Time (s)
500	.106
1000	.804
1500	2.698
2000	6.382
2500	12.459
3000	21.519
3500	34.162

The running times are plotted. The cubic nature of the curve is clearly visible.



Exercise

Create a Makefile and use the Makefile to compile the program listed above. Run the program using the `time` command on Linux to see some running times. Collect running times on two different machines. Write a brief report indicating your results and listing the machine specifications. List reasons why one machine is faster than the other.

To find out machine specs on Linux, check out the `lspci` and `lscpu` commands, and check out the `/proc` filesystem. Check the man pages and/or Google for information on how to use/interpret these resources

2.2 Sum3 Simple Implementation (Serial)

For an input array of arbitrary integers (that is, we make no assumptions as to the range of the integers), *Sum3* can be solved in $O(n^2)$ time using relatively simple algorithms.

The purpose of this chapter is to emphasize the point that parallelism is not always the best solution to a problem. Sometimes, making algorithmic improvements (resulting in better theoretical complexity) can provide better speedups than parallelism, especially if you don't have the hardware resources to take advantage of a parallel algorithm.

2.2.1 Using a Hash Table

The easiest approach to code is to take advantage of hash tables. Hash tables are implemented in many languages, so they are cheap from a coding perspective, and they offer $O(1)$ time for inserting an element and determining if an element is in the hash table ... under ideal circumstances. The approach is to:

1. Load all numbers in a hash table
2. Generate all possible pairs of numbers
3. Check if the negation of the sum of each pair of numbers is in the hash table.

However, this approach suffers from some drawbacks. First, hash table operations can degrade to linear time under certain conditions (although this can be managed rather easily). Second, care must be taken to ensure that unique triples are identified. For example, if the input array was `-2, 0, 4`, one must ensure that `-2+-2+4 = 0` does not get counted as a valid triple (since -2 only appears once in the input).

2.2.2 Using a Sorted Input Array

A slightly more complicated $O(n^2)$ algorithm requires that the input array be sorted. The algorithm contains a loop that will traverse the input array from first to last.

Assume a sorted input array `array` containing `n` elements. We will first find all triples that add up to 0 and involve the first element in `array`. We set `a=array[0]`. A triple is then formed with `b=array[0+1]` and `c=array[n-1]`. If `a+b+c > 0`, then we need to reduce the sum to get it closer to `0`, so we set `c=array[n-2]` (decrease the largest number in the triple to get a smaller sum). If `a+b+c < 0`, then we need to increase the sum to get it closer to 0, so we set `b=array[0+2]` (increase the smaller number to get a larger sum). if `a+b+c = 0`, we have a triple that adds up to 0, and we either set `c=array[n-2]` or `b=array[0+2]` in order to test the next triple (in the code below, we set `c=array[n-2]`). This process continues until `b` and `c` refer to the same array element. At the end of this process, all possible triples involving the first array element have been computed.

To compute all triples involving the first array element, we took a pair of numbers from the array. After examining the pair, one number was excluded from further consideration in future triples involving the first array element. Therefore, for `n` array elements, we will construct `n-2` triples due to the following: the first array element is used in every triple, leaving us to create pairs of the remaining `n-1` elements; the first pair uses two elements and eliminates one from further consideration; all following pairs will reuse 1 element from the previous pair, and eliminate one element from consideration. Thus, computing all triples involving a single element requires $O(n)$ time. Repeating this for `n` elements in the input array requires $O(n^2)$ time. A C++ implementation of this algorithm follows:

```
#include <iostream>
#include <sstream>

using namespace std;
int getRandInt( int *dataVec, int dataSize )
{
        // load up a vector with random integers
        int num;
```

```
        for( int i = 0; i < dataSize; i++ ) {
                // integers will be 1-100
                num = rand() % 100+1;
                if( rand( ) % 2 == 0 ) {
                        // make some integers negative
                        num *= -1;
                }
                dataVec[i] = num;
        }
}


int main(int argc, char * argv[] )
{
        int dataSize = 0;

        if( argc < 2 ) {
                std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
                exit( -1 );
        }
        {
                std::stringstream ss1;
                ss1 << argv[1];
                ss1 >> dataSize;
        }
        if( argc >= 3 ) {
                std::stringstream ss1;
                int seed;
                ss1 << argv[2];
                ss1 >> seed;
                srand( seed );
        }
        else {
                srand( 0 );
        }

        // create a data vector
        int *data = new int[ dataSize ];

        // load it up with random data
        getRandInt( data, dataSize );
        // sort the data
        sort( data, data + dataSize );

        // do the Sum3 computation.  O(n^2)
        int count = 0;
        int a,b,c, sum; // array elements
        int j,k;   // array indices
        for (int i = 0; i < dataSize-2; i++){
                a = data[i];
                j = i+1;
                k = dataSize-1;
                while( j < k ) {
                        b = data[j];
                        c = data[k];
                        sum = a+b+c;
                        if( sum == 0 ){
                                cerr << a << " " << b << " " << c << endl;
```

```
                                count++;
                        }
                        if( sum < 0 ){
                                j++;
                        }
                        else {
                                k--;
                        }
                }
        }
        cout<< count <<endl;
```

}

One pitfall in this algorithm is that duplicate values cause problems.

Example

Consider the input string `array = [-4, 1, 1, 3, 3]`. There are 4 triples that sum to 0:

```
array[0] + array[1] + array[3] = 0
array[0] + array[1] + array[4] = 0
array[0] + array[2] + array[3] = 0
array[0] + array[2] + array[4] = 0
```

However, the above code will only find 2 triples. The algorithm progresses as follows, with items being examined in bold:

-4, **1**, 1, 3, **3** sum = 0, so index **k** is decremented
-4, **1**, 1, **3**, 3 sum = 0, so index **k** is decremented
-4, **1**, **1**, 3, 3 sum = -2, so index **i** is incremented, resulting in j == k and the while loop exits

Effectively, the algorithm only finds triples involving the first **1**, and none involving the second **1**.

In fact, duplicates cause 3 problems that must be addressed:

1. 3 or more 0s will independently sum to 0. In fact, for `n > 2` 0s, there will be `n choose 3` triples that sum to 0.
2. A sequence of repeating numbers exists such that a pair of those numbers along with a third forms a triple that sums to 0. For example: ‘‘ -4, 1, 2, 2, 2‘‘ contains 3 triples that sum to 0. When such a sequence is detected with length `n > 2`, there will be `n choose 2` triples that sum to 0. Note that a correct handling of this case can also handle the case where 3 or more 0s exist.
3. The situation in the above example when two sequences of repeating numbers exist such that the repeated element of each sequence along with a third number not equal to those elements sum to 0. For sequences of non-equal numbers `x` and `y` with respective lengths `m >= 1` and `n >= 1`, such that there exists a number `z | x+y+z = 0`, there will be `x*y` triples that sum to 0 for each individual copy of `z`.

The following program handles duplicates correctly. **Problem 1** and **Problem 2** are handled in a single conditional, the first `if` statement in the `while` loop. Note that when such a situation occurs, we have effectively found the point

at which `j` and `k` converge, and so we break out of the while loop. **Problem 3** is handled in the second `if` statement in the `while` loop; the modification of `j` and `k` are again handled specially in this situation (since they may be increased or decreased by more than 1). The third `if` statement in the `while` loop handles the situation in which duplicates do not occur. The final `else` block in the `while` loop handles the normal increment of `j` or decrement of `k` when a triple with a non-zero sum is visited. The complete solution for handling duplicates is:

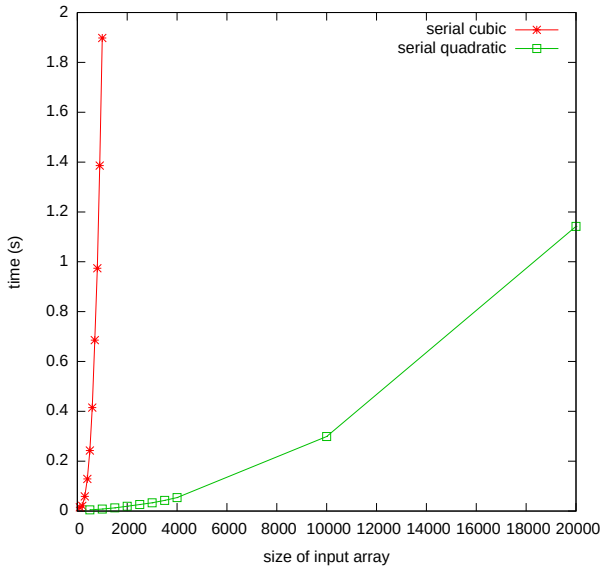
```
#include <iostream>
#include <sstream>

using namespace std;
int getRandInt( int *dataVec, int dataSize )
{
        // load up a vector with random integers
        int num;
        for( int i = 0; i < dataSize; i++ ) {
                // integers will be 1-100
                num = rand() % 100 +1;
                if( rand( ) % 2 == 0 ) {
                        // make some integers negative
                        num *= -1;
                }
                dataVec[i] = num;
        }
}


int main(int argc, char * argv[] )
{
        int dataSize = 0;

        if( argc < 2 ) {
                std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
                exit( -1 );
        }
        {
                std::stringstream ss1;
                ss1 << argv[1];
                ss1 >> dataSize;
        }
        if( argc >= 3 ) {
                std::stringstream ss1;
                int seed;
                ss1 << argv[2];
                ss1 >> seed;
                srand( seed );
        }
        else {
                srand( 0 );
        }

        // create a data vector
        int *data = new int[ dataSize ];

        // load it up with random data
        getRandInt( data, dataSize );
        // sort the data
        sort( data, data + dataSize );

```

```
        // do the Sum3 computation.  O(n^2)
        int count = 0;
        int a,b,c, sum; // array elements
        int j,k;    // array indices
        for (int i = 0; i < dataSize-2; i++){
                a = data[i];
                j = i+1;
                k = dataSize-1;
                while( j < k ) {
                        b = data[j];
                        c = data[k];
                        sum = a+b+c;
                        if( sum == 0 && b == c ) {
                                // case where b == c.  ie, -10 + 5 + 5
                                // or where a == b == c == 0
                                int num = k-j+1;
                                count += (num*(num-1))/2;
                                break;
                        }
                        else if( sum == 0 && (data[j+1] == b || data[k-1] == c )){
                                // case where there are multiple copies of b or c
                                // find out how many b's and c's there are
                                int startj = j;
                                while( data[j+1] == b ) j++;
                                int startk = k;
                                while( data[k-1] == c ) k--;
                                count += (j-startj+1) * (startk-k+1);
                                j++;
                                k--;
                        }
                        else if( sum == 0 ){
                                // normal case
                                count++;
                                j++;
                        } else {
                                // if sum is not 0, increment j or k
                                if( sum < 0 ) j++;
                                else k--;
                        }
                }
        }
        cout<< count <<endl;
}
```

Here are some running times for the previous program:

Array Size	Time (s)
500	.005
1000	.008
1500	.013
2000	.019
2500	.026
3000	.033
3500	.043
4000	.054
10000	.299
20000	1.142

Here we show the graph of running times for the cubic and quadratic versions of the algorithm. Notice the vast difference in running times for the two programs. Again, the cubic and quadratic nature of the algorithms are visible.



Exercise

Create a Makefile and use the Makefile to compile the program listed above. Run the program using the `time` command on Linux to see some running times. Collect running times on two different machines. Write a brief report indicating your results and listing the machine specifications. List reasons why one machine is faster than the other.

To find out machine specs on Linux, check out the `lspci` and `lscpu` commands, and check out the `/proc` filesystem. Check the man pages and/or Google for information on how to use/interpret these resources.

2.3 Sum3 Parallel Implementation with Pthreads and OpenMP

At this point we have seen two possible algorithms and implementations for the Sum3 problem

1. A $O(n^3)$ algorithm that is extremely easy to implement, but that suffers from poor execution time.
2. A $O(n^2)$ algorithm that is much faster in terms of execution time, but that requires care to handle sequences of repeating values correctly.

The question now is whether or not we can do better. It turns out that algorithmically, we cannot...at least not yet; there are currently no known algorithms to solve the Sum3 problem faster than $O(n^2)$ time. In fact, a class of problems exists called Sum3 hard problems *[GOM1995CG]*. Any problem that is constant time reducible to a Sum3 problem is Sum3-hard; the implication being that a sub-quadratic time solution to any problem in the Sum3-hard class provides a sub-quadratic time solution to Sum3.

2.3.1 Introducing Parallelism with Multiple Threads

In terms of running time, we can do better with the Sum3 problem. One way to improve running time is to utilize a multi-core CPU with a multi-threaded program. The general approach is to divide the work that must be completed among multiple cores. Ideally, if we evenly divide the work among two processors, the program should run twice as fast. Another way to express this is to say that the program should have a 2x **speedup** over the serial version.

The concept of speedup is a useful measure to gauge the effectiveness of a particular optimization, and is defined as follows:

Definition

Let t_{old} be the running time of a program and t_{new} be the running time of the same program that has been optimized. The **speedup** of the new program is equal to the running time of the old program divided by the running time of the new program:

speedup $= t_{old}/t_{new}$

Design of a Multi-Threaded Algorithm

Designing a multi-threaded program, or a parallel algorithm for that matter, requires a different thought process than defining a serial program. When defining a serial algorithm, the focus is usually to determine the steps that must be completed in succession to achieve an answer. When developing a multi-threaded program, one must think in terms of how the work can be divided among threads. It is sometimes useful to begin with a serial program and try to modify it to divide work, and other times it is easier to simply start from scratch.

Concept

How should the work be divided among threads?

In order to effectively parallelize work, each thread must be able to do a portion of the work. Ideally, each thread will take roughly the same amount of time to execute; it is generally undesirable for a bunch of threads to complete their work quickly and sit around waiting on a single, slow thread to finish.

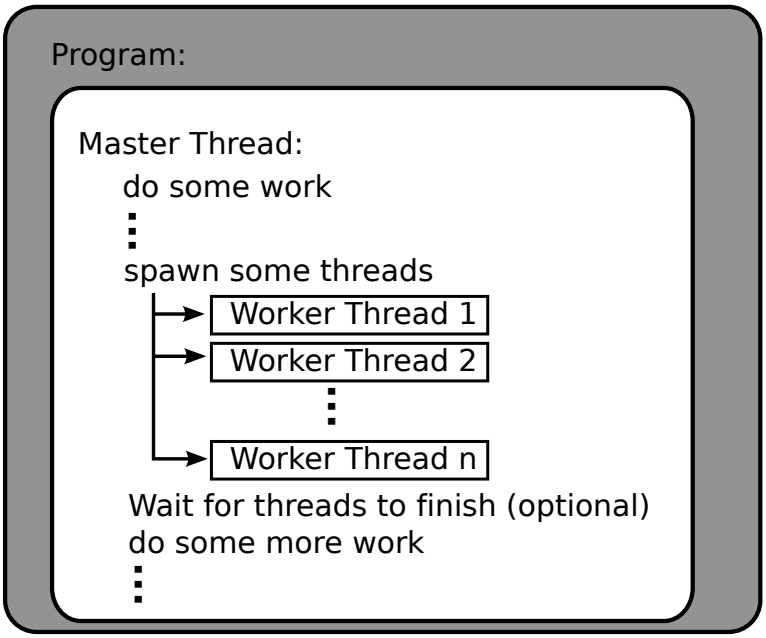
Lets take the cubic version of the Sum3 algorithm as an example (the core of the program is listed below). The main work of the program consists of the double nested loops that generate all possible triples of numbers from the input array. A reasonable method of dividing the work is to make 2 threads such that each thread generates roughly half of the triples that must be tested. We can achieve this by modifying the outer loop such that one thread will only compute triples in which the first number in the triple comes from the first half the array, and the second thread will compute triples in which the first number comes from the second half of the array. Such a modification requires minor changes to the core of the program.

```
#include <iostream>
using namespace std;

int main( )
{
        int dataSize = 5;
        int* data = new int[ dataSize ];
        data[0] = -1;
        data[1] = -2;
        data[2] = 0;
        data[3] = 2;
        data[4] = 3;
        // do the naive Sum3 computation.  O(n^3)
        int count = 0;
        for (int i = 0; i < dataSize-2; i++)
                for (int j = i+1; j < dataSize-1; j++)
                        for (int k = j+1; k < dataSize; k++)
                                if (data[i] + data[j] + data[k] == 0)
                                        count++;

```

```
21          cout<< count <<endl;
22
23  }
```

Cubic Sum3 Using Pthreads

Pthreads are a low level threading mechanism that have been around for a long time. Pthreads are provided by a C library, and as such, they do things in a very old-school, C way of thinking. The basic idea is that the programmer defines a function, and then tells the pthread library to create a thread to run that function. Therefore, we have a concept of a master thread, which is the thread of the program that is started when the program is initially executed, and the concept of worker threads, which are spawned by the master thread. Typically, the master thread will spawn some worker threads, and then wait for them to complete before moving on, as illustrated in the following:



In order to achieve a pthread version of the program, we must first put the code that each thread will execute into a function. We will then call a pthread library call and tell it to use that function as the code the thread will run. The pthread library will then launch the thread. Our master thread can then launch additional threads, do other work, or wait on the launched threads to finish executing. Here are the function calls we will use... you should check your system's man pages for the most up to date information.

Pthread Library Calls

```
int
pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr,
void *(*start_routine)(void *), void *restrict arg);
```

Create a thread. The first argument will be assigned a pointer to a thread handle that is created. The second argument is a pointer to a thread attributes struct that can define attributes of the thread; we will simply pass a NULL to use defaults. The third argument is the function pointer to the function that this thread will execute. The final pointer is a pointer to a single argument that will be passed to the function. Note, to use the argument, the function must type cast it to the appropriate type. Also, if more than 1 arguments are required, you must pack them into a **struct**.

Returns a 0 on success, a non-zero error code on failure

```
int
pthread_join(pthread_t thread, void **value_ptr);
```

Join a thread. The join command will cause the calling thread to wait until the thread identified by the thread handle in the first argument terminates. We will pass NULL as the second argument.

Returns a 0 on success, a non–zero error code on failure

Now that we have the ability to create and join threads, we can create a simple pthread program. The following example shows how to create threads to do some work, and pass arguments to them. Remember that pthreads require the function that will be executed as a thread to take a single argument; so we must wrap all the arguments we want to pass to the function into a single struct. We define the struct on line 23, and instantiate two instances of the struct on line 87. Because a pointer to the struct is passed to our `partial3SumThread` function as a `void *`, we must cast the pointer back to the correct type on line 33 before we can use it in the function.

We want to pass the input array to each thread, so we can simply pass it to the function.

Note

The function launched as a thread by the pthread library *must* take a single argument of type void pointer. This argument must be type-cast to a void to pass it into the function, and type-cast back to its original type within the function. Of course, this breaks type checking for the compiler, so be careful and double check the types yourself!

Here is the code:

```
1  #include <iostream>
2  #include <sstream>
3
4  using namespace std;
5
6  int getRandInt( int *dataVec, int dataSize )
7  {
8          // load up a vector with random integers
9          int num;
10         for( int i = 0; i < dataSize; i++ ) {
11                 // integers will be 1-100
12                 num = rand() % 100 +1;
13                 if( rand( ) % 2 == 0 ) {
14                         // make some integers negative
15                         num *= -1;
16                 }
17                 dataVec[i] = num;
18         }
19 }
20
21 // define a struct to pass information to the threads
22 struct threadInfo{
23         int myID;
24         int *dataPtr;
25         int dataSize;
26         int count;
27 };
28
29
30 void* partial3SumThread( void* arg ) {
31         // type cast the argument back to a struct
32         threadInfo * myInfo = static_cast<threadInfo*>( arg );
```

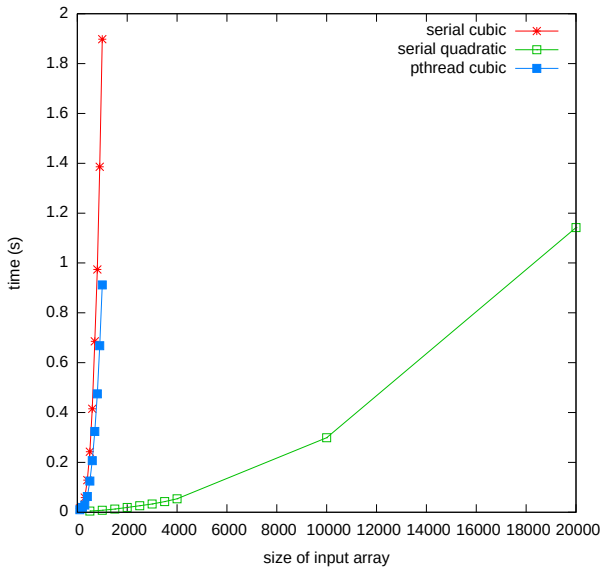
```
33
34
35          // each thread only works on half the array in the outer loop
36          // compute the bounds based on the ID we assigned each thread.
37          // remember, we only have 2 threads in this case, so we will hard code a 2
38          int start = ((myInfo->dataSize / 2) * myInfo->myID );
39          int stop = ((myInfo->dataSize / 2)* (myInfo->myID+1));
40          if( myInfo->myID == 1 )
41                  stop =myInfo->dataSize-2;
42
43          // do the naive Sum3 computation.  O(n^3)
44          for (int i = start; i < stop; i++)
45                  for (int j = i+1; j < myInfo->dataSize-1; j++)
46                          for (int k = j+1; k < myInfo->dataSize; k++)
47                                  if (myInfo->dataPtr[i] + myInfo->dataPtr[j] + myInfo->dataPtr[k] == (
48                                          myInfo->count++;
49                                  }
50  }
51
52  int main(int argc, char * argv[] )
53  {
54          int dataSize = 0;
55
56          if( argc < 2 ) {
57                  std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
58                  exit( -1 );
59          }
60          {
61                  std::stringstream ss1;
62                  ss1 << argv[1];
63                  ss1 >> dataSize;
64          }
65          if( argc >= 3 ) {
66                  std::stringstream ss1;
67                  int seed;
68                  ss1 << argv[2];
69                  ss1 >> seed;
70                  srand( seed );
71          }
72          else {
73                  srand( 0 );
74          }
75
76          // create a data vector
77          int *data = new int[ dataSize ];
78
79          // load it up with random data
80          getRandInt( data, dataSize );
81
82          // allocate thread handles
83          pthread_t       worker1tid, worker2tid;
84
85          // allocate and set up structs for 2 threads
86          threadInfo info1, info2;
87          info1.myID = 0;
88          info1.dataPtr = data;
89          info1.dataSize = dataSize;
90          info1.count = 0;
```

```
91          info2.myID = 1;
92          info2.dataPtr = data;
93          info2.dataSize = dataSize;
94          info2.count = 0;
95
96          // allocate space for a return value
97          int returnVal;
98          // call the worker threads
99          if (  returnVal = pthread_create( &worker1tid, NULL, partial3SumThread, &info1 ) ) {
100                 cerr<<  "pthread_create 1: "<< returnVal <<endl;
101                 exit( 1 );
102         }
103         if (  returnVal = pthread_create( &worker2tid, NULL, partial3SumThread, &info2 ) ) {
104                 cerr<<  "pthread_create 2: "<< returnVal <<endl;
105                 exit( 1 );
106         }
107         // now wait on the threads to finish
108         if ( returnVal = pthread_join( worker1tid, NULL ) ) {
109                 cerr<<  "pthread_join 1: "<< returnVal <<endl;
110                 exit( 1 );
111         }
112
113         if ( returnVal = pthread_join( worker2tid, NULL ) ) {
114                 cerr<<  "pthread_join 2: "<< returnVal <<endl;
115                 exit( 1 );
116         }
117
118         cout<< info1.count + info2.count <<endl;
119 }
```

The program has a relatively simple structure; the core of the Sum3 computation is done in the `partial3SumThread` function. 2 instances of the function are launched, and we assign the first instance an ID of 0, and the second instance an ID of 1. These thread IDs are manually assigned and stored in a `threadInfo` struct that we create. The structs are created and initialized on lines 87-95. Based on the thread's ID, we determine which half of the array a thread should use in the outer loop of the Sum3 computation (lines 39-42). A thread is launched on line 99 and line 103. Once all the threads are launched, the main thread must wait for them to complete before printing the result, so the main thread calls one join function for each thread (lines 108 and 113). Once all worker threads have joined, we print the result.

Here are some running times on a machine with a 2 core cpu, and the results graphed along with the serial and serial quadratic versions of the algorithm:

Array Size	Time (s)
100	.01
200	.019
300	.029
400	.063
500	.125
600	.207
700	.324
800	.475
900	.668
1000	.912



The pthreads version is clearly faster. As expected, the version with two threads runs roughly twice as fast as the version with 1 thread. For an array of 500, the speedup is 1.9, for an array of 1000, the speedup is 2.08. A speedup of more than 2 is likely due to timing granularity. The Linux `time` command was used, which is not the most accurate time keeping method. Other considerations should have an impact. For example, the two threads could have an effect on cache, causing a greater amount of cache hits or misses than the single threaded version. Cache misses cause the processor to stall, wasting time. More cache hits means the processor stalls less, and does the work in a shorter amount of time.

Exercise

The pthreads version of the Sum3 problem is hard coded to handle 2 threads. Convert the program such that it will take the number of threads to use as a command line argument, and then generate that many threads. Hint: you will have an easier time of it if you use vectors to hold all thread information. For example, you will have a vector of `threadInfo` structs, one for each thread.

Thread Communication

In the pthreads version of Sum3, very little thread communication took place. In fact, the only thread communication that happened was the master thread provided some data to the worker threads in the form of a function argument. Usually, threads will communicate with each other through **shared memory**. The key to understanding shared memory is understanding scoping rules. Basically, a thread will have access to all memory locations that are in scope for that thread, and all memory locations accessible through pointers that are in scope for that thread. This means that if a variable has global scope, then ALL threads will be able to read and write to it. This means, two threads can communicate by reading and writing to such a variable. In our program above, we declared two instances of the `threadInfo` struct in the `main()` function (line 87); those instances are `info1, info2`. Those structs now exist in memory, but the names `info1, info2` exist in scope for the `main()` function. We store information that the threads need in those structs. To provide access to that information, we pass the pointer to one of those structs to each thread during the `pthread_create` call. This means that a thread can access its struct through that pointer. Essentially, the master thread has communicated with the worker threads, letting them know their thread ID, the input array, etc.

Note that in the program, only 1 copy of the `data` array exists. It is created in the `main()` function. Pointers to that array are stored in the `threadInfo` structs, so that each thread can access the same array. This does not cause any problems because both threads only read the array, they are not making changes to it. Also note that each struct has

its own memory location to store a `count`. Thus, each thread is keeping track of its own `count`. This is why all the `count` values from the `threadInfo` structs must be summed in line 119.

Exercise

Change the pthreads Sum3 program such that both threads access the same memory location for keeping track of their `count` variable. Don't use any other pthread library features. Run the program multiple times. What happens to the result value? why?

2.3.2 Doing Things a Little Differently with OpenMP

Pthreads is just one method to create multi-threaded programs. Most languages have some built-in mechanism or library to launch threads. Another mechanism in C/C++ is OpenMP. One drawback to pthreads is that although the core Sum3 computation changed very little in the pthread program, the program did require some somewhat significant structural changes. Recall that we had to put our computation into a function, set up a struct to pass data to the function, introduce create and join function calls, etc. Furthermore, we only implemented 2 threads; implementing more threads requires even more work. So, there is some programming overhead to converting a serial program to a parallel program using pthreads.

OpenMP takes a compiler-based approach to threading. Instead of inserting function calls and re-organizing code to create threads, OpenMP requires you to place compiler directives near portions of code that will be run in parallel so the compiler can generate a team of threads. These directives are known as **pragmas**. One of the goals of OpenMP is to provide a mechanism whereby serial code can be parallelized using multiple threads with only minor code modifications. The result is that you can write a serial program, debug it while it is serial (a much easier task than debugging parallel code), and then simply add a pragma to achieve parallelism using multiple threads.

As an example, we will write a OpenMP version of the Sum3 problem with a similar structure as the pthread version: we will create 2 threads, and each thread will only iterate over a portion of the outer loop. We will need the following OpenMP pragmas and functions:

OpenMP Interface

```
1 #pragma omp parallel
2 {
3         // some code
4 }
```

Will create a team of n threads where n is the number of computational cores available on the computer on which this code is executed. Each thread will execute the code in the code block. No code will be executed beyond the code block until all threads have joined.

```
1 int omp_get_thread_num();
```

Returns the unique thread identifier of the thread that executes this function. If only a single thread exists, the function always returns 0. If multiple threads exist, the function returns a number in the range from `0` to `omp_get_num_threads()-1`, inclusive.

```
1 int omp_get_num_threads();
```

Returns the number of threads in the current team. If only 1 thread exists (the serial portion of the code), it returns 1.

```
1 void omp_set_num_threads( int num_threads );
```

Sets the default number of threads to create in subsequent parallel sections (for example, sections defined by `#pragma omp parallel`).

```
g++ -fopenmp source.cpp
```

remember to use the OpenMP compiler flag to compile programs using OpenMP.

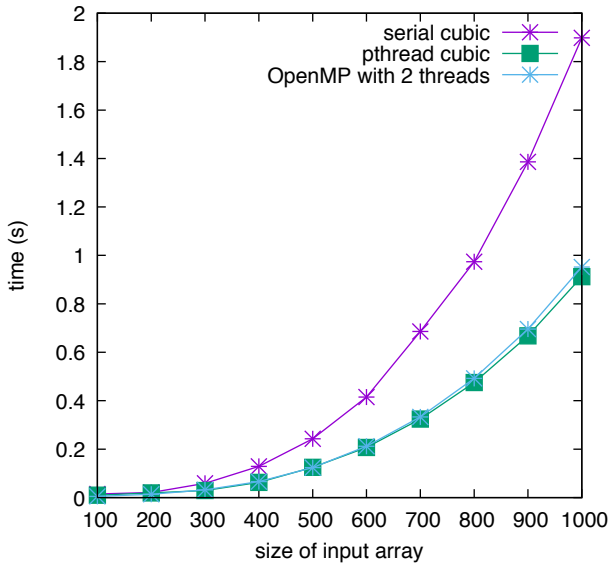
With these pragmas and functions, we can duplicate the pthreads version of the program by forcing 2 threads to operate on the data. We will call `omp_set_num_threads(2)` (line 57, below) to force 2 threads, then break up the work similarly to what we did before. Much like pthreads, OpenMP threads will communicate through memory locations; again, scoping rules apply: (1) any variable whose scope is external to a parallel section will be shared among all threads, and (2) all threads will have their own private copy of any variable declared within the scope of a parallel section. So, we will define an integer for each thread to keep track of the number of triples summing to 0 that it sees external to the parallel section, so we can add those counts together at the end (lines 54-55). Recall in the pthread code, we had to explicitly set thread identifiers to 0 and 1; OpenMP will do this for us and each thread can find its ID using the `omp_get_thread_num()` function (line 62):

```
#include <iostream>
#include <sstream>
#include <omp.h>

using namespace std;

int getRandInt( int *dataVec, int dataSize )
{
        // load up a vector with random integers
        int num;
        for( int i = 0; i < dataSize; i++ ) {
                // integers will be 1-100
                num = rand() % 100 +1;
                if( rand( ) % 2 == 0 ) {
                        // make some integers negative
                        num *= -1;
                }
                dataVec[i] = num;
        }
}


int main(int argc, char * argv[] )
{
        int dataSize = 0;

        if( argc < 2 ) {
                std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
                exit( -1 );
        }
        {
                std::stringstream ss1;
                ss1 << argv[1];
                ss1 >> dataSize;
        }
        if( argc >= 3 ) {
                std::stringstream ss1;
                int seed;
                ss1 << argv[2];
                ss1 >> seed;
                srand( seed );
        }
        else {
```

```
                srand( 0 );
        }

        // create a data vector
        int *data = new int[ dataSize ];

        // load it up with random data
        getRandInt( data, dataSize );

        // do the naive Sum3 computation.  O(n^3)
        int count1 = 0;
        int count2 = 0;


        omp_set_num_threads( 2 );
#pragma omp parallel
        {
                int count = 0;
                int myID = omp_get_thread_num();
                // each thread only works on half the array in the outer loop
                // compute the bounds based on the ID we assigned each thread.
                // remember, we only have 2 threads in this case, so we will hard code a 2
                int start = (( dataSize / 2) * myID );
                int stop = ((dataSize / 2)* (myID+1));
                if( myID == 1 )
                        stop =dataSize-2;
                for (int i = start; i < stop; i++)
                        for (int j = i+1; j < dataSize-1; j++)
                                for (int k = j+1; k < dataSize; k++)
                                        if (data[i] + data[j] + data[k] == 0){
                                                count++;
                                        }
                if( myID == 0 )
                        count1 = count;
                else
                        count2 = count;
        }
        cout<< count1 + count2 <<endl;
}
```

Now, some running times. Again, speedup is roughly 2 for the OpenMP version, but the OpenMP version required much less code reorganization to implement. The running times are a few thousandths of a second higher than the pthread versions. This is due to a different implementation of threading. In OpenMP, the compiler implements the threading, rather than OS system calls. Because of this, the threads get compiled a little differently. Also, different OpenMP compilers will have slightly different performance results.

Array Size	Time (s)
100	.005
200	.014
300	.032
400	.066
500	.124
600	.212
700	.333
800	.492
900	.696
1000	.952



Exercise

Compile the serial cubic version of the program and the OpenMP version with compiler optimizations turned on (for example, use the -O3 with the gcc or g++ compilers). How do the running times compare then?

Creating a Lot of Threads

Because OpenMP is integrated into the compiler, the overhead of creating threads is small; thus, it is possible to create many more threads than available processor cores. An important concept of OpenMP is the idea of worker threads: on a computer with 2 cores, the number of worker threads typically defaults to 2 (there are system parameters that control the default settings). Even if many threads are created, only 2 will be able to run at any given time. Therefore, it is acceptable, although not always optimal, to create many more threads than there are processors available. One easy way to create a lot of threads is through the `omp parallel for` pragma. This pragma must be placed directly before a `for` loop. OpenMP will then treat each iteration of the loop as a block of work that can be assigned to a worker thread; therefore, if you have 10 worker threads and 10 loop iteration, all iterations can be run in parallel on individual worker threads. If there are 10 worker threads and 11 iterations, one worker thread will effectively run two loop iterations, but those iterations are still treated as separate blocks of work; one thread simply picks up the extra block of work when it finishes its first block. To be safe, make sure that the `for` loop contains the declaration of the looping variable (this will be more clear when we talk about shared memory below)!

One problem with generating lots of threads is that we also need to then generate lots of `count` variables (lines 54-55 above). One alternative is to create a single global count variable, but then make sure that only 1 thread accesses it at a time. To ensure that only 1 thread accesses a variable at a time, we put that variable access in a **critical section** using an OpenMP `omp_critical` pragma. Anything in a code block directly following an `omp_critical` pragma is guaranteed to be accessed by exactly 1 thread at a time. Using these directives, our multi-threaded code looks very similar to our original code:

```
1  #include <iostream>
2  #include <sstream>
3  #include <omp.h>
4
5  using namespace std;
6
7  int getRandInt( int * dataVec, int dataSize )
```

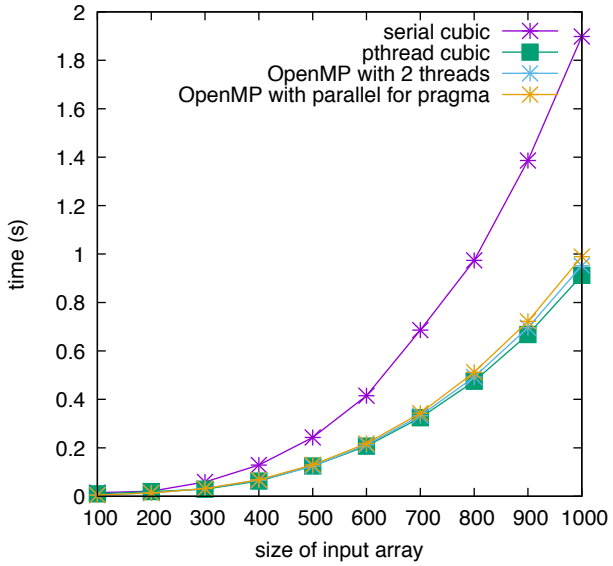
```
{
    // load up a vector with random integers
    int num;
    for( int i = 0; i < dataSize; i++ ) {
        // integers will be 1-100
        num = rand() % 100 +1;
        if( rand( ) % 2 == 0 ) {
            // make some integers negative
            num = num * -1;
        }
        dataVec[i] = num;
    }
}


int main(int argc, char * argv[] )
{
    int dataSize = 0;

    if( argc < 2 ) {
        std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
        exit( -1 );
    }
    {
        std::stringstream ss1;
        ss1 << argv[1];
        ss1 >> dataSize;
    }
    if( argc >= 3 ) {
        std::stringstream ss1;
        int seed;
        ss1 << argv[2];
        ss1 >> seed;
        srand( seed );
    }
    else {
        srand( 0 );
    }

    // create a data vector
    int * data = new int[ dataSize ];

    // load it up with random data
    getRandInt( data, dataSize );

    int count = 0;
    // do the naive Sum3 computation.  O(n^3)
    #pragma omp parallel for
    for (int i = 0; i < dataSize-2; i++)
        for (int j = i+1; j < dataSize-1; j++)
            for (int k = j+1; k < dataSize; k++)
                if (data[i] + data[j] + data[k] == 0){
                    #pragma omp critical
                    {
                        count++;
                    }
    }

```

```
66      cout<< count <<endl;
67  }
```

The one drawback of the critical section is that execution may serialize on it. Therefore, if every thread enters the critical section in every execution of the loop, then the program will behave much like a serial program in terms of running time. Therefore, placement of the critical section is important! Note that we placed it inside the `if` statement. Every triple must be tested to see if it sums to 0, but only a small portion of the triples actually sum to 0. If we placed the `if` statement in the critical section, we would get serial behavior. If every triple summed to 0 (an array of lots of 0s), we would also get serial behavior (regardless of the placement of the critical section with respect to the `if` statement). Thus, the previous method will have more reliable speedups in these edge cases, but this method will get speedups for input that we are likely to see, and requires only a few lines of modification to the code, and no logic changes!

Here are the running times. Note that the critical section does have an impact, but it is not too bad. A speedup of 1.9 vs 1.9 for the pthreads version at an array size of 500 (the same!), and a speedup of 1.9 vs 2.1 for the pthreads version at an array size of 1000:

Array Size	Time (s)
100	.005
200	.014
300	.032
400	.068
500	.130
600	.218
700	.343
800	.512
900	.722
1000	.989



Exercise

Change the OpenMP program using a critical section to serialize in 2 ways. First, use an input array of all 0's. Then move the critical section to contain the `if` statement. Compare running times to the serial version of the algorithm (the cubic one), and give a list of reasons as to why one is faster/slower than the other.

Reductions

OpenMP has a lot of other options, constructs, and functionality. Check the official documentation for more details *[OpenMPDoc]*. One of the advantages of OpenMP is that many of these constructs make it very easy to achieve tasks that one has to do manually with pthreads. A good example of this is the `reduction`.

A reduction simply means to combine many values into a single value using a specified operator. For example, we can reduce an array of numbers to a single sum of all the numbers in the array (by adding them up). Alternatively, we could reduce an array to its max or min value, by searching for the largest or smallest value, respectively, in the array. We can use a reduction to completely get rid of the critical section in the previous version of the code.

When declaring an `omp parallel for` pragma, we can identify values that will be reduced after all threads have executed. Each thread will then make its own local copy of any variable specified in a reduction, and OpenMP will automatically perform the specified reduction of all of those local variables into a single value. The reduction operation is also specified.

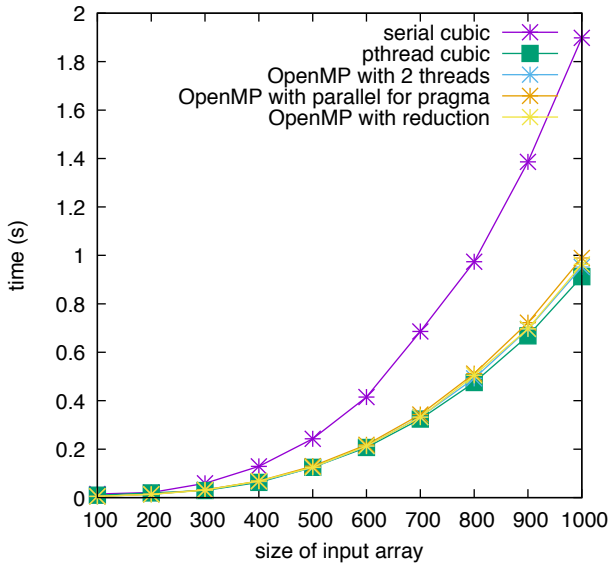
For example, to get rid of the critical section above, we should tell OpenMP to perform a reduction on `count`. Thus, every thread will get its own copy of count. We will specify a sum reduction, so that at the end of thread execution, all the local versions of the `count` variable get summed into a single variable. The result is that we only have to add a single line of code to achieve parallel implementation of Sum3. Line 55 specifies that a sum reduction on the variable `count` will be computed:

```
1  #include <iostream>
2  #include <sstream>
3  #include <omp.h>
4
5  using namespace std;
6
7  int getRandInt( int *dataVec, int dataSize )
8  {
9          // load up a vector with random integers
10         int num;
11         for( int i = 0; i < dataSize; i++ ) {
12                 // integers will be 1-100
13                 num = rand() % 100 +1;
14                 if( rand( ) % 2 == 0 ) {
15                         // make some integers negative
16                         num *= -1;
17                 }
18                 dataVec[i] = num;
19         }
20 }
21
22
23 int main(int argc, char * argv[] )
24 {
25         int dataSize = 0;
26
27         if( argc < 2 ) {
28                 std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
29                 exit( -1 );
30         }
31         {
32                 std::stringstream ss1;
33                 ss1 << argv[1];
34                 ss1 >> dataSize;
35         }
36         if( argc >= 3 ) {
37                 std::stringstream ss1;
```

```
                int seed;
                ss1 << argv[2];
                ss1 >> seed;
                srand( seed );
        }
        else {
                srand( 0 );
        }

        // create a data vector
        int *data = new int[ dataSize ];

        // load it up with random data
        getRandInt( data, dataSize );

        int count = 0;
        // do the naive Sum3 computation.  O(n^3)
#pragma omp parallel for reduction(+:count)
        for (int i = 0; i < dataSize-2; i++)
                for (int j = i+1; j < dataSize-1; j++)
                        for (int k = j+1; k < dataSize; k++)
                                if (data[i] + data[j] + data[k] == 0){
                                        count++;
                                }
        cout<< count <<endl;
}
```

And finally, here are some running times:

Array Size	Time (s)
100	.005
200	.013
300	.031
400	.068
500	.125
600	.213
700	.332
800	.505
900	.697
1000	.963



The running times are a bit slower than the pthreads version, and a bit faster than the OpenMP version with a critical section. However, remember that the pthreads version is hard coded to 2 threads, and the programmer essentially must implement their own reduction if you want to use more threads. This version required adding 1 pragma to an otherwise unchanged serial implementation and achieved threading that will automatically scale to the number of available processors with NO extra work! This code is easy to debug (you simply comment out the pragma and debug in serial, or compile without the OpenMP flag and the pragmas should be ignored), uses advanced features, and is easily adaptable to multiple hardware configurations. This should convince you of why OpenMP is a popular choice for threading.

Exercise

Look at the scheduling clauses for OpenMP (the wiki has a concise description *[OpenMPWiki]*). Try dynamic scheduling with various chunk sizes on the above program. Report the effects on execution time, and describe why those effects are occurring.

Convert the $O(n^2)$ version of the algorithm to a parallel algorithm using OpenMP. Try to get the fastest time for 100,000 numbers. Make sure to use a computer with multiple cores (and hyperthreading turned off!)

This material is based upon work supported by the National Science Foundation under Grant Numbers 1258604 and 1347089. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

CHAPTER

THREE

MASSIVE PARALLELISM WITH CUDA

Warning: The Cuda code in these examples that use the *atomicAdd()* function must be compiled with compiler flags that support atomic functions.
In general, architecture version 2.0 introduced a lot of very useful functionalility, and has been around sufficiently long that one can usually assume their card supports this level of functionality. Technically, atomic operations using integer arguments was first available in leve 1.1. Thus, to compile these programs, the command will look like:
nvcc -arch=sm_20 SOURCE.cu -o EXENAME (for architecture version 2.0)
nvcc -arch=sm_11 SOURCE.cu -o EXENAME (for architecture version 1.1)

Creator Name: Mark McKenney

Content Title: Introduction to the MapReduce Model of Parallelism

Learning Objectives:

1. Describe the general architecture of a GPU.
2. Explain the major, conceptual, architectural differences between a CPU and a GPU processor.
3. Define the concept of a *thread block* in Cuda.
4. Identify portions of an algorithm that are good candidates for parallelization.
5. Design algorithms with threads that take advantage of GPU architectures.
6. Explain the concept of thread divergence.
7. Describe the basic outline of a Cuda program.
8. Explain the aspects of thread communication that are unique to the GPU.
9. Describe the concepts of *grid*, *grid dimensions*, *blocks*, and *threads* and how they relate.
10. Use error checking on the GPU.
11. Describe the memory hierarchy of the GPU.
12. Use shared memory to optimize kernel.

Background Knowlege

1. Ability to write, compile, and execute a C/C++ program.
2. Ability to debug simple programs.

Resources Needed

1. Access to a computer with a Cuda enabled GPU.
2. Cuda drivers and the Cuda development kit (nvcc) installed on that computer

Work Mode

The module is self-contained. One may work through the document alone, or instructors can prepare lectures based on the information and samples.

In this document, we introduce GPU-style parallelism using Cuda and explore some solutions to the 3sum problem utilizing a GPU co-processor. We begin with a brief introduction to GPU architecture. Using GPU architecture as a guide, we introduce GPU programming using Cuda. Cuda is an extension of the C programming language that is created and maintained by Nvidia. Finally, we develop some solutions to 3sum using Cuda.

3.1 An Introduction to GPU Architectures

Section Goals

1. Describe the big picture concepts related to GPU architectures.
2. Describe the role of an *accelerator* in a computer system.
3. Show how a piece of work can be split in a large number of threads.

The traditional computer architecture that we have considered in the multi-threaded discussions consists of a processor and a memory that communicate over a bus. The processor may have multiple cores, or may be multiple physical processors, but we have thus far treated the computing portion of this model as a single, logical processor, possibly with multiple cores, as shown in the following figure.

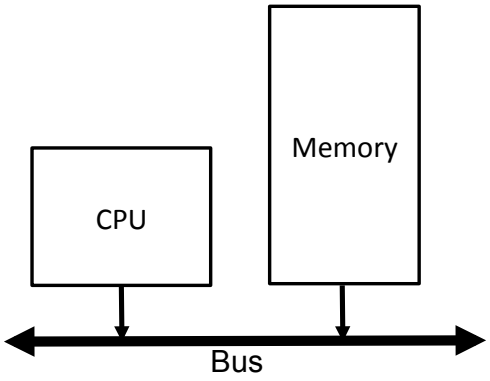


Fig. 3.1: A general architecture in which a CPU and memory communicate over a high speed bus.

The model of GPU programming we will explore using Cuda adds a second processing unit to the architectural model that attaches to the bus. Generally, this processing unit is may be referred to as an **accelerator**, and is not necessarily a GPU, although we will assume that it is. The GPU is a self-contained unit that has its own processing units and its **own** memory. The entire unit connects to the CPU bus as shown:

The fact that the GPU has its own memory is significant; the GPU must have all data that it needs for its computations in its own memory! In fact, transfering data from the system memory to the GPU memory must be explicitly done by the programmer. Also, transferring memory between system and GPU memory can be a performance bottleneck, and must be managed carefully. Having a separate memory that must be managed by the programmer is the first major difference that we encounter when using accelerators.

Concept

The GPU has its own memory. In order for the GPU to process a piece of data, it must be explicitly transferred from the main memory to the GPU memory.

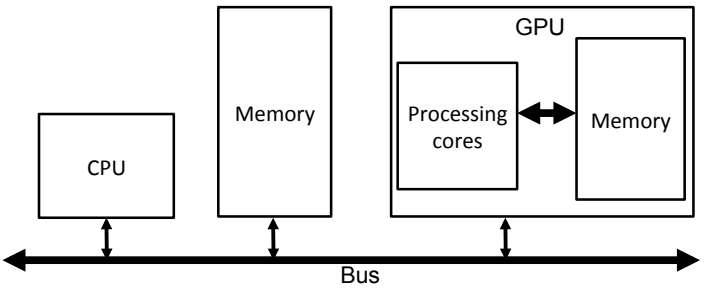


Fig. 3.2: An architecture in which a GPU is attached to the bus along with the CPU and Memory. Note that the GPU has its own internal computation and memory units that communicate over an internal bus.

We denote the GPU's memory: **GPU memory**.

We denote the main memory: **Host memory** or **CPU memory**.

The GPU memory is very similar to CPU memory: it is simply a set of storage locations with incrementally increasing byte addresses.

The GPU processing units are very different from a CPU's processing units. In general, computer CPU cores are meant to be general purpose processors that are quite complex and can execute a large variety of algorithms efficiently. In a sense, each individual core on a desktop processor is designed to execute a single thread at a time (with exceptions for hyperthreading) and do all sorts of fancy tricks to re-order instructions, predict which way branches go, and identify and resolve data dependencies. A single GPU processing unit, on the other hand, is a simplistic processing core without the bells and whistles. Desktop CPUs take the path in which a few, very complex and powerful cores provide good performance: CPU cores these days have in the ballpark of 4-12 cores. The GPU takes the path in which each individual processor core is very simple, but there are very many cores: A GPU will have on the order of 2000 computational cores.

GPU processing cores are not independent, as are CPU cores. Groups of GPU cores are linked such that in that group of cores, all cores are executing the same instruction in the same clock cycle. Therefore, GPUs are very good at processing items in arrays in which each array element must have the same operations performed it. For example, if you wanted to increment every number in an array, the GPU could do this well since all cores would be performing the same operation (incrementing a value) on a different data element (each core gets one value from the array and increments it). You probably see that while a CPU core is good at large variety of algorithms, on certain types of algorithms work well on a GPU; particularly, algorithms based on array or matrix manipulations tend to be a good fit to GPUs.

Concept

So at this point, we know that:

1. GPUs have a large number of rather simplistic processing elements.
2. GPUs have their own memory and can only operate on data that is explicitly transferred into GPU memory
3. GPU processing units are organized into groups such that each core in a group executes the same instruction in the same clock cycle (typically on a different piece of data in each processing unit).

These three concepts form the foundation for understanding how to effectively program a GPU (or other similar accelerator) for maximum performance.

Finally, lets look at performance. The architectures of CPUs and GPUs are very different, and thus they have very different performance characteristics. Intel haswell core processors have about .2 TFLOPS of theoretical performance. Thats about 200 trillion floating point operations per second. An Nvidia K40 has about 1.5 TFLOPS of double

precision performance and 4.3 TFLOPS of single precision performance. As you can see, the GPU architecture is capable for achieving an order of magnitude more performance than a desktop processor. However, a GPU can't get that performance for every algorithm! In fact, a single threaded algorithm or an algorithm that does not make use of the GPU architecture effectively will actually run slower on a GPU than on the CPU. Thus, in order to take advantage of the computational power of the GPU, we need to develop algorithms so that they can take advantage of GPU architecture, and divide the work in the algorithms effectively among processing core groups in the GPU.

3.2 The CUDA Computational Model

Section Goals:

1. Explain the major, conceptual, architectural differences between a CPU and a GPU processor.
2. Define the concept of a *thread block* in Cuda.

In the previous section, we talked about the differences in the processing cores of CPU processors vs GPU processors. In this section, we expand on that discussion by looking in a more detailed way at the architectural differences between CPU and GPU processor organization, and then seeing how GPU algorithms need to be structured to fit that architecture. Note that we used the term **processing units** to describe a single core on a processor in the previous section. From this point forward, we will create a few definitions to clear up the terminology:

Processor A full processor with possibly multiple computational cores.

Core A single computational core on a processor.

Streaming processor In effect, a single core on a GPU processor. Although the concepts are not quite equivalent, conceptually they are similar.

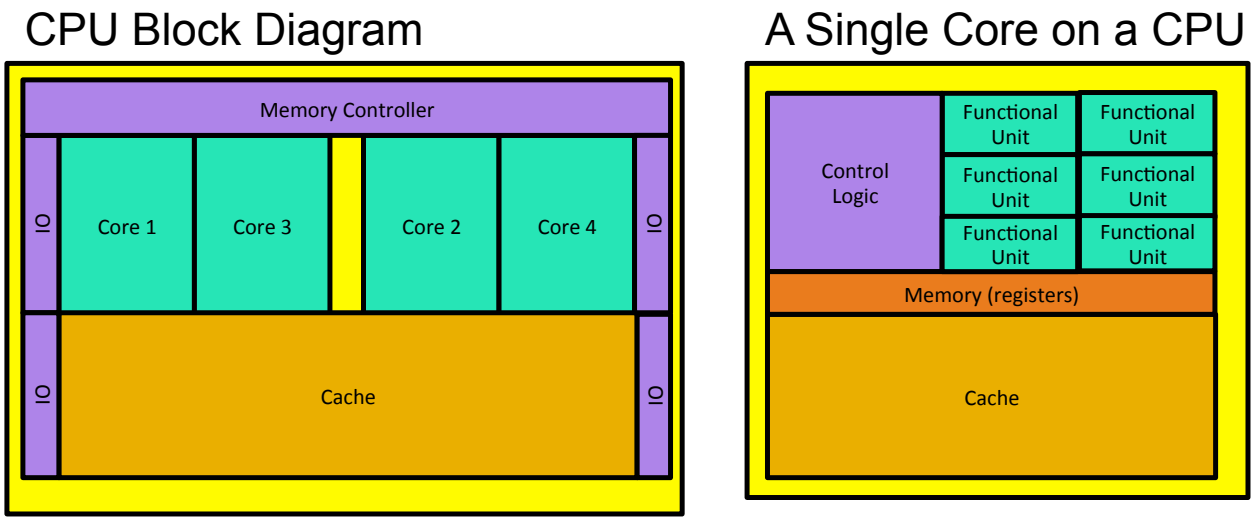


Fig. 3.3: A typical CPU block diagram of both an entire CPU and a single core on a CPU.

A typical CPU block diagram looks like diagram in Fig. 3.3. As was mentioned previously, you often see on the order of 4-12 individual cores, and a decent chunk of cache memory. Note that each core is relatively large; this is because each core is complex, and generally only handles a single thread at a time (for more information on the cores, check out the following terms: processors pipelining, superscalar, and hyperthreading). Inside each core is more memory, some control logic, and then some functional units. A functional unit might be an ALU (add/subtract numbers, etc.), a floating point multiplier, a memory interface unit, etc. Basically, each functional unit provides some sort of basic functionality. However, because the core is meant for one thread, there is not a huge number of functional units, and it there is a lot of space dedicated to control and memory in an attempt to some complicated things to speed up thread

execution (for more information look up the terms: data hazards, branch prediction, stall cycles, memory stall cycles).

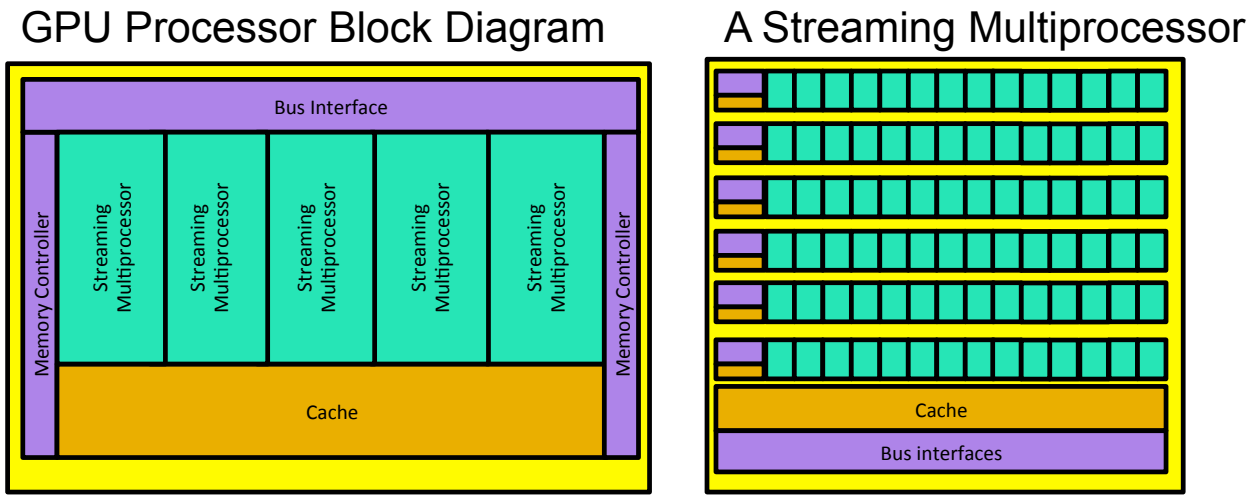


Fig. 3.4: A typical GPU block diagram of both an entire GPU processor and a single streaming multiprocessor (essentially a core).

A typical GPU block diagram is depicted in Fig. 3.4. Instead of a series of cores, a gpu processor has series of *streaming multiprocessors*, what Nvidia calls SMXs [1]. Looking at the diagram of a single streaming multiprocessor, the differences from a CPU core jump out. The SMX contains a bunch of functional units; however, a series of functional units is controlled by a single control unit with its own *small amount* of cache. Essentially, every functional unit controlled by a single control unit will be executing the same operations (add, subtract, multiply, etc.), during the same clock cycle. Even though they all do the same thing in the same clock cycle, they are all doing it to different pieces of data (again, remember the example about incrementing every element in an array). So, instead of a CPU core executing a single thread at a time, the SMX actually executed *many* threads simultaneously. However, those threads need to be executing the same instruction in the same clock cycles.

From a purely visual perspective, one notices that the SMX processor devotes a significantly larger amount of area to functional units, i.e., to computation, as compared to the CPU core. Because a CPU core executes 1 thread at a time, it needs lots of cache in order hold a lot of memory on chip in an effort to reduce the amount of times a piece of data must be retrieved from main memory. Basically, the CPU is much faster than main memory access, so any time a CPU needs data from memory, it has to stall and wait on the data to arrive. By keeping a lot of memory on chip, a CPU can (and often does) avoid these stalls. The SMX has very little cache. Instead, it takes the approach that since it designed to execute many threads, any thread that is stalled waiting on a memory request is scheduled out of the CPU, and another thread that is currently scheduled out can be put into the CPU and begin executing. Therefore, memory stalls are *hidden* because the threads that are waiting for memory access get to wait while other threads jump into the processor and execute. Note that this model assumes there **lots** of threads. Ideally, there are more threads than functional units available; really you want a number of threads that is a multiple of the number of functional units available.

Concept

In a GPU processor, threads that are stalled on memory requests are scheduled out of execution, and threads that are ready to execute are put into the functional units. Therefore. This approach to hiding memory stalls works best when

[1] In reality, Cuda GPUs have a higher level grouping called Graphics Processing Clusters (GPCs) which contain multiple SMXs and some other units, such as a raster engine. However, in terms of general purpose computing on the GPU at the level covered in this document, we are mainly interested in the SMXs.

there are more threads than functional units avaialable. In general, you want to have many more threads than functional units available.

Recall the OpenMP model of threading in which it is very easy to create a large number of threads. The GPU processor is an ideal candidate for this concept of creating many of threadsi (although we will not use OpenMP to do it since OpenMP is specific to CPUs. Check out OpenAcc for the OpenMP counterpart for the GPU).

Because many functional units are controlled by a single control unit, it organize threads into **blocks** such that each block can be sent to a group of functional units controlled by a single control unit. In essence, a block is just a group of threads that have identical code (so they can all do the same thing in the same clock cycle). Really, we will usually create a bunch of blocks that all do the same thing. A group of blocks will be sent to a SMX, and each block will then get dispatched to a group of functional units. In keeping with the idea of creating lots of threads, it is OK to send more blocks to an SMX than it has functional units to handle. The blocks will then get switched in and out of functional units to hide memory stalls.

Block A group of threads that all contain the same code. Each thread will execute the same code (usually) on a different data element. Again, think about array elements.

Thread A thread is a single thread of execution. Threads are organized into blocks.

Concept

To effectively use GPU hardware, we need an algorithm that can be designed to create many **threads** grouped into multiple **blocks** such that each thread contains identical code, but will execute on indepenent data.

Algorithms based on arrays and matrices tend to fit this extremely well.

3.3 Creating Algorithms that Divide Work into Many Threads

Section Goals:

1. Identify portions of an algorithm that are good candidates for parallelization.
2. Design algorithms with threads that take advantage of GPU architectures.

With any multi-threaded/parallel/distributed program to be successful, we need to design an algorithm with a few properties:

1. The work must be be able to be divided into independent units that can be computed (mostly) individually.
2. If threads need to communicate, the communication must be designed so that it occurs efficiently.
3. If threads need to collaborate, the collaboration must be designed so that threads are not spending too much time waiting on values produced by other threads.

Communication and collaboration are similar. The difference tends to arise in distributed programming scenarios with lots of threads running on different machines. In such scenarios, one must consider network communication delays, proximity of servers on a network, etc. In such cases, communication and collaboration manifest separately. In your typical multithreaded application that runs on a single computer, they can often be treated as one concept. Similarly in the case of your typical GPU application.

3.3.1 Monte Carlo Algorithm to Approximate Pi

Lets take an example. One fun algorithm to play with is the **Monte Carlo Algorithm for Approximating** π. A *Monte Carlo* algorithm is essentially a statistical simulation that uses sequences of random numbers as data. In order to

approximate pi, we will take advantage of some geometric constructs, and the generate a bunch of random data points.

The basic idea for this algorithm comes from the fact that the ration of the area of a circle with radius r to the area of a square with edge length $2r$ is equal to 4π:

$$\frac{Area\ of\ a\ Circle}{Area\ of\ a\ Square} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4} = 0.78539816$$

The basic idea is that if we can compute the ration of the areas, lets call it α, then we just multiply it by 4 to get pi. So, how do we get the ratio?

One way to compute the ratio α is to just count up the number of points that lie inside the circle, and divide it by the number of points that lie inside the square. However, there are an infinite number of points in both the circle and the square, and we must do this on a finite computer. So, lets do a Monte Carlo simulation. Our approach is to simply generate n random points inside the square, and count up the number of points that happen to fall only in the circle m. The ration $m/n \approx \alpha$. An illustration of this approach is shown in Fig. 3.5

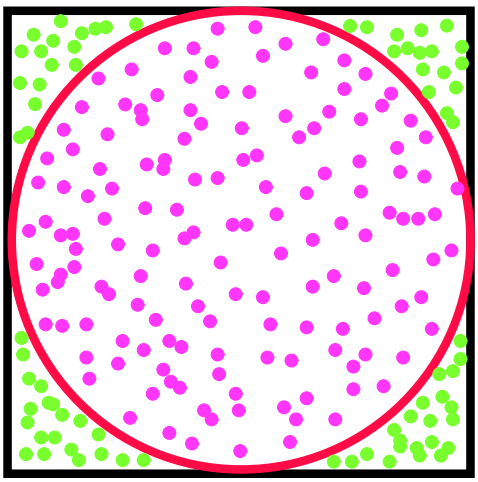


Fig. 3.5: An illustration of generating random points in a unit square with a circle inscribed in it.

We can also observe that we don't necessarily need the entire square and circle, we can get the same ratio by just considering a single quadrant of the square (Fig. 3.6). This simplifies the code a bit.

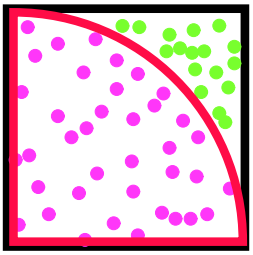


Fig. 3.6: Using just one quadrant of the square preserves the desired ratio, and ultimately makes the code more simple.

The algorithm to do this computation looks like the following:

```
var count = 0
Repeat X times:
    x = random value between 0 and 1
    y = random value between 0 and 1
    if x*x+y*y <= 1:
```

```
        count = count + 1
piApprox = 4 * (count/X)
```

The algorithm is rather simple. The next step is to just figure out the best way to assign work to threads. Clearly, the majority of the work occurs in the `for` loop. In fact, just about all the work occurs in the `for` loop. So, we should try to divide up that work. Notice that each iteration of the loop is rather independent. Each iteration simply gets some random values and does some calculations. Iterations do write to the same `count` variable, so they must be careful about race conditions, but this is the only problem. So, a natural division of work is that each thread will execute some number of loop iterations. The next question is: how many threads? Recall that the GPU is very good at creating lots of threads, and in fact, uses the fact that it is good at having lots of threads to hide memory latencies. So, lets say that we will create a bunch of threads. Each thread can then do some work, lets say 1000 iterations of the loop, and keep track of its own local copy of the count variable. At the end, we will have to sum all the local count variables into a single, final count value. At this point, we have not introduced many of the coding constructs needed to create a working Cuda program, but we go and show a Cuda implementation of the π approximation algorithm. Some of the particulars are discussed below the algorithm, but this should give you a general idea of how a Cuda program will look:

```
#include <sstream>
#include <iomanip>
#include <cuda.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <curand.h>
#include <curand_kernel.h>

#include <iostream>

using namespace std;

#define NUM_POINTS_PER_THREAD 1000

__global__ void kernel_initializeRand( curandState * randomGeneratorStateArray, unsigned long seed,
                                       int totalNumThreads )
{
    int id =  (blockIdx.x * blockDim.x) + threadIdx.x;
    if( id >= totalNumThreads ){
        return;
    }
    curand_init ( seed, id, 0, &randomGeneratorStateArray[id] );
}

__global__ void kernel_generatePoints( curandState* globalState, int* counts, int totalNumThreads )
{
    // calculate which thread is being executed
    int index =  (blockIdx.x * blockDim.x) + threadIdx.x;
    // create vars
    float x,y;

    // make sure we don't go off the end of the counts array
    if( index >= totalNumThreads ){
        return;
    }

    // get the state for the random number generator
    // of this thread
    curandState localState = globalState[index];

```

```
    // generate the points, see if they fall in the unit circle
    for(int i = 0; i < NUM_POINTS_PER_THREAD; i++ )
    {
        x = curand_uniform( &localState );
        y = curand_uniform( &localState );
        if( x*x+y*y <= 1 )
            counts[ index ] ++;
    }
    globalState[index] = localState;
}

int main( int argc, char** argv)
{
    if( argc < 2 ){
        std::cerr << "usage: " << argv[0] << " [number of threads to use] " << endl
        << "This program approximates pi using a Monte Carlo approximation and the "<<
        << "'square inside a circle' method." <<endl
        << "Each thread will generate 1000 points"<< endl;
        exit(0);
    }
    int numThreads;
    {
        stringstream ss1( argv[1] );
        ss1 >> numThreads;
    }

    // Step 1: Figure out how many thread blocks we need.
    dim3 threadsPerBlock( 1024, 1, 1);
    dim3 numberofBlocks( (numThreads + threadsPerBlock.x-1)/threadsPerBlock.x,1,1);
    // Step 2: initialize the random number generator on the GPU. Each thread will be generating
    // its own sequence of random numbers.  The random number generator actually does not generate
    // random numbers.  Instead, it generates a sequence of numbers that behave as if they are rando
    // Basically, if the random number generator is started at the same point in the sequence (and
    // with the same SEED value) for multiple runs, it will generate the SAME sequence of numbers!

    // Since each thread generates its OWN pseudorandom sequence, each thread needs to keep track of
    // the STATE of its random number generator.  Thus, we create a curandState struct for each threa
    // that will be generating random numbers.  When we make a call to the random numebr generator, u
    // simply pass the state values to it. This is the standard way of allocating memory on the GPU
    curandState* devRandomGeneratorStateArray;
    cudaMalloc ( &devRandomGeneratorStateArray, numThreads*sizeof( curandState ) );


    // Step 3: we need to create some vectors.  This way uses the thrust libraries to
    // allocate GPU memory and perform transfers from CPU to GPU memory.
    //  Its a little easier than the tradtional way.
    // create a vector to hold the counts for each thread
    thrust::host_vector<int> hostCounts(numThreads, 0);
    // transfer that vector to the GPU
    thrust::device_vector<int> deviceCounts( hostCounts );
    // get a pointer to that vector so we can pass it to the kernel.
    int* dcountsptr = thrust::raw_pointer_cast( &deviceCounts[0] );


    // Step 4: Initialize the random number generator states.  Again, each thread needs its own
    // state.  We will seed each of the states based on the current time.
    // launch a bunch of copies of the kernel (1 for every thread)
    kernel_initializeRand <<< numberofBlocks, threadsPerBlock >>> ( devRandomGeneratorStateArray,
```

```
99                                                                    time(NULL), numThreads );
100
101     // Step 5: Generate the points and get counts for how many fall in the unit circle
102     kernel_generatePoints <<< numberofBlocks, threadsPerBlock >>> ( devRandomGeneratorStateArray,
103                                                                    dcountsptr, numThreads );
104
105     // Step 6:  Each thread generated its own count.  We need to add them all up
106     // the easy way is to use a thrust reduction on our thrust vector
107     // this will occur on the GPU since we use the vector on the GPU
108     // we will perform a recution on the entire array, beginning with a sum value initialized to 0,
109     // and perform a `plus` operation.  This will just add up all the numbers in the vector.
110     int sum = thrust::reduce(deviceCounts.begin(), deviceCounts.end(), 0, thrust::plus<int>());
111
112     std::cout << "Our approximation of pi =" <<std::setprecision(10)
113               << (float(sum)/(numThreads*NUM_POINTS_PER_THREAD))*4 << std::endl;
114     return 0;
115 }
```

In Cuda terminology, a **kernel** is a function that will be run on a GPU. Our code has two kernels. The first simply initializes the states of the random number generators for each thread that will run (because we have to generate random points). The second is actually the body of the `for` loop in our algorithm. The **__global__** keyword simply tells the compiler that the function can be called from code running on the CPU or from code running on the GPU.

Recall that we said we would generate a bunch of threads, and each thread would run through about 1,000 iterations. In the `kernel_generatePoints` kernel, there is a `for` loop that runs 1,000 times. Remember, the kernels are constructed so that a single thread executes a single kernel. Recall that in a group of functional units in a SMX, all functional units are executing the same code in the same clock cycle, but are working on different data. Therefore, A bunch of threads execute the same kernel (so they have the same code), and in this case, each thread generates its own random data.

Another issue to think about is the `count` variables. Remember, we said that each thread would have its own version of the count variable. Our approach in this code was to create a vector of integers such that each thread would only use 1 element in the vector as its own count variable. At the end, we have to sum all the counts for each individual vector into the total count.

Because each thread runs a single kernel, we must tell our program how many threads to create. The program takes a command line parameter that indicates the number of threads to create. Cuda will simply create as many threads as you tell it to; threads will automatically be numbered consecutively beginning with number 0. Cuda wants to group threads into **blocks**. A block just holds a group of threads, and the blocks are used for scheduling threads to groups of functional units in an SMX. On lines 68 and 69, we tell Cuda how many threads we want in a block, and then we compute how many blocks we will need based on the input parameter that user specifies to indicate the number of threads to create. This information is used at the kernel launch calls on lines 98 and 102 to tell Cuda how many kernels will be launched (1 kernel per thread). Typically, the first item of business in a kernel is to determine the thread's ID number (lines 18 and 28). All of this will be explained in more detail later.

Again, remember that we have to allocate memory both on the GPU and on the CPU. The GPU works on items in GPU memory, so that data must explicitly be allocated. In our code, we use two mechanisms. The first is the traditional Cuda way of allocating memory, using `cudamalloc()`. Lines 80 and 81 allocate some memory to hold the random number generator states for each thread that will be launched. The second mechanism is to allocate vectors using the **thrust library**. Thrust is the equivalent of the *standard template library* for Cuda. It allows much more concise syntax for allocating GPU memory and transferring data between the GPU and CPU.

We can run the program multiple times and see that the approximation changes slightly, but does a pretty good job. Also, you can play with the number of threads launched to see its effect on accuracy and run time.

```
$ nvcc -arch=sm_20 piGen.cu -o pigen

$ ./pigen 20
```

```
Our approximation of pi =3.130199909

$ ./pigen 200
Our approximation of pi =3.139499903

$ ./pigen 2000
Our approximation of pi =3.143307924

$ ./pigen 20000
Our approximation of pi =3.141334295

$ ./pigen 200000
Our approximation of pi =3.141306162

$ ./pigen 2000000
Our approximation of pi =3.141551018
```

As the number of generated points becomes larger, the accuracy of the approximation tends to improve. Note that in the last run we used $2,000,000$ threads, each running $1,000$ iterations. That is $2,000,000,000$ points! Its a lot of points.

Excercise

Create a file containing the program above. Get it to compile, and run it with different amounts of threads to get a feel for its performance. Try a few things like altering the number of iterations in a single kernel call. Try changing the number of threads per block.

Note

Notice the vast majority of the code in the above example is simply setting up memory and kernel calls. The actual work (done in the `for` loop in the kernel) is only about 5 lines of code.

3.4 Dividing Work Among Cuda Threads

Section Goals:

1. Design algorithms with threads that take advantage of GPU architectures.
2. Explain the concept of thread divergence.
3. Explain the aspects of thread communication that are unique to the GPU.

As we saw in the last example, one key to effectively programming any parallel system is to divide the work well. However, we need to keep a couple of things in mind with the GPU.

3.4.1 Dividing the Work

Often, GPU kernels tend to be rather short in terms of lines of code. As kernels grow in complexity, it becomes more difficult to ensure that you will achieve maximum performance. There are a few reasons for this. One is that each kernel has a limited amount of registers that can be allocated to it in hardware. If this number is exceeded, the program will crash. Another is that you don't want to have a situation where 1 thread in a block has a lot of work to do, and

the others are finished. The GPU gets its power from massive parallelism. Any the amount of work among threads in a block is unbalanced, you will see sub-optimal performance.

Concept

Kernels tend to be rather short and to the point. One wants to ensure the amount of work being done by threads in a block is roughly equal.

3.4.2 Branch Diversion

Some kernels contain `if` statements. By definition, an `if` statement forces the thread of execution to either skip over a code block, or execute it. One cannot expect that ALL threads in a block will ALWAYS follow the same thread of execution through `if` statements. For example, in lines 45 and 46 in the example above, some threads will create a random point that falls in the circle, and others will not. Recall that all threads executing in a group of functional units must execute the same instruction in the same clock cycle. So, what happens on lines 46 and 47?

When two threads executing in a group of functional units do encounter a situation where one thread enters the block controlled by the `if` ' statement, and the other does not, we say the threads have **diverged**. Basically, the thread for which the `if` statement evaluated to `False` must simply wait until the other thread executes the code block controlled by the `if statement`. This means that one thread stalls while the other executes. This is called **thread divergence**

Thread divergence can occur when conditional statements and loops exist in kernels. Remember, any time thread divergence occurs, some threads must simply stall and burn clock cycles. If all threads in a block follow the same path of execution in a kernel, then no thread divergence, and thus no stalls, occur. This is another reason why kernels tend to be short and simple.

3.4.3 Thread Communication

Sometimes individual threads must communicate. This typically occurs through shared memory, but GPUs have the problem that very large numbers of threads are often created. This situation means that just acquiring locks on memory addresses may often lead to sub-optimal performance.

The rules of shared memory in Cuda are rather intuitive. Any variables declared within a kernel are local in scope to that kernel, and each individual thread will have its own, private memory location to store that data. Any pointers to memory passed in to a kernel as an argument points to a memory location that was allocated outside the scope of the kernel, and thus, all threads will have a pointer to that one, shared location in memory.

If threads must write to a shared global memory address, they often do it through **atomic** operations, such as **atomicAdd**. The atomic operators guarantee that race conditions will not occur, but often at the cost of speed.

In our example above, each thread had its own copy of `count`. Our approach was to create a vector in global memory that all threads could access, but only allow each thread to access 1 array element. That way, all threads wrote to independent memory locations, and race conditions were not a problem. The only problem was at the end, we had an array of counts for each thread, and not a total. So, we had to do one more step and sum the contents of the array. The practice of reducing an array of values down to a single value is called a **reduction**. Luckily, the Thrust library provides many common reductions, such as sum, max, min, etc., and we just used that.

Concept

A **ruduction** occurs when you take an array of values, and perform some operation on it to reduce it to a singel value (or sometimes, a smaller set of values). Common examples are:

1. Compute the sum of an integer array.
2. Find the maximum value in an array.

3. Find the minimum value in an array.

Because communicating through shared memory is often problematic in a GPU because of the large numbers of threads, reductions are very common. This is another reason why kernels tend to be short and simple. Instead of one big kernel performing an entire algorithm, algorithms are often broken into smaller pieces so that reductions or other data transformations can occur between steps in the algorithm.

3.5 An Example: Sum3

Section Goals:

1. Describe the concepts of *grid*, *grid dimensions*, *blocks*, and *threads* and how they relate.
2. Use error checking on the GPU.

The program to approximate π gives a brief example of a Cuda program to give the reader a general idea of the structure of a Cuda program and the issues that tend to arise. In this section, we develop a Cuda program incrementally to explain all the details. We will continue the example of the **Sum3** problem.

Recall that the Sum3 problem takes an array, and returns the number of triples in that array that sum to the value 0. The following example illustrates:

Example

The result of computing the Sum3 algorithm on the following list of numbers is 2:

$-1, -2, 0, 2, 3$

Since:

$-1 + -2 + 3 = 0$

$-2 + 0 + 2 = 0$

Again, the simplest way to compute the result is to use a double nested *for* loop to simply generate all possible triples and test them. Clearly, this solution has a time complexity of $O(n^3)$. For example:

```
#include <iostream>
using namespace std;

int main( )
{
        int dataSize = 5;
        int* data = new int[ dataSize ];
        data[0] = -1;
        data[1] = -2;
        data[2] = 0;
        data[3] = 2;
        data[4] = 3;
        // do the naive Sum3 computation.  O(n^3)
        int count = 0;
        for (int i = 0; i < dataSize-2; i++)
                for (int j = i+1; j < dataSize-1; j++)
                        for (int k = j+1; k < dataSize; k++)
                                if (data[i] + data[j] + data[k] == 0)
                                        count++;

```

```
21          cout<< count <<endl;
22
23  }
```

So, lets break this down into a simple Cuda program.

3.5.1 Step 1: Divide the Work Among Threads

The first step is to figure out how much work each thread will do. We will use our knowledge of the GPU architecture to pick an amount of work that is relatively manageable and straightforward in an attempt to reduce thread divergence and unnecessary complexity. Once the program works, we will worry about increasing the complexity to achieve higher performance.

In the basic CPU version of the program, the purpose of the loops is to simply generate a triple of values from the array. Remember, the GPU is very good at handling large numbers of threads. So, one way to go about this is to generate a single thread for every triple that must be generated. Each thread just tests to see if its triple sums to 0. However, that is probably a little bit of overkill in terms of thread creation since this it will generate $O(n^3)$ threads. Instead, lets have each thread do a bit more work. Lets have each thread contain a single *for* loop: thus, a thread will be generated with 2 values of a triple fixed, and it will generate all possible triples that contain those two fixed values:

Example

If we want each thread to have 2 fixed values, then for the situation depicted in the previous exmaple code that uses the following array:

$-1, -2, 0, 2, 3$

We will generate threads that do the folllowing:

Thread	Fixed values	Triples tested in the thread
1	-1, -2	(-1,-2,0), (-1,-2,2), (-1,-2,3)
2	-1, 0	(-1, 0,2), (-1, 0,3)
3	-1, 2	(-1, 2,3)
4	-2, 0	(-2, 0,2), (-2, 0,3)
5	-2, 2	(-2, 2,3)
6	0, 2	(0, 2,3)

Therefore, we need a kernel that has two fixed array indexes, and loops over the array for the third. Cuda provides a mechanism by which we can assign identifiers to threads. The basic idea is that thread numbers start at 0, and increment up to the number of threads we wish to generate. This makes it very easy to assign a particular array location to a particular thread for computation. Furthermore, we can structure these thread IDs so that they are 2 or 3 dimensional so that they correspond to 2D array (matrix) locations or 3D array (matrix) locations. Cuda supplies some special, built-in variables that allow us to access the ID of a particular in a kernel. Remember, many threads execute the same kernel, but with different thread IDs, allowing us to easily assign different array locations to be processed by different threads.

The special built in variables that provide access to a thread's ID are:

threadIdx A struct containing 3 integers: *threadIdx.x*, *threadIdx.y*, and *threadIdx.z*

blockIdx A struct containing 3 integers: *blockIdx.x*, *blockIdx.y*, and *blockIdx.z*

blockDim A struct containing 3 integers: *blockDim.x*, *blockDim.y*, and *blockDim.z*

Instead of simply numbering threads from 0 up to the total number of threads, threads are divided into blocks. Each block begins numbering threads at 0. To get the threads actual number (as if the numbers just incremented across the

entire range of threads), you must add the *threadIdx* value to the *blockIdx* * *blockDim* values. The following example depicts this concept visually

---

Example

Assume we have a 2D array (a matrix) with 10 rows and 11 columns. We want to create kernel threads such that each thread access a single cell in the matrix. The most straightforward to do this is for each thread to have an ID that corresponds to a matrix cell. So, we will create threads with 2-dimensional IDs (a row and column value).

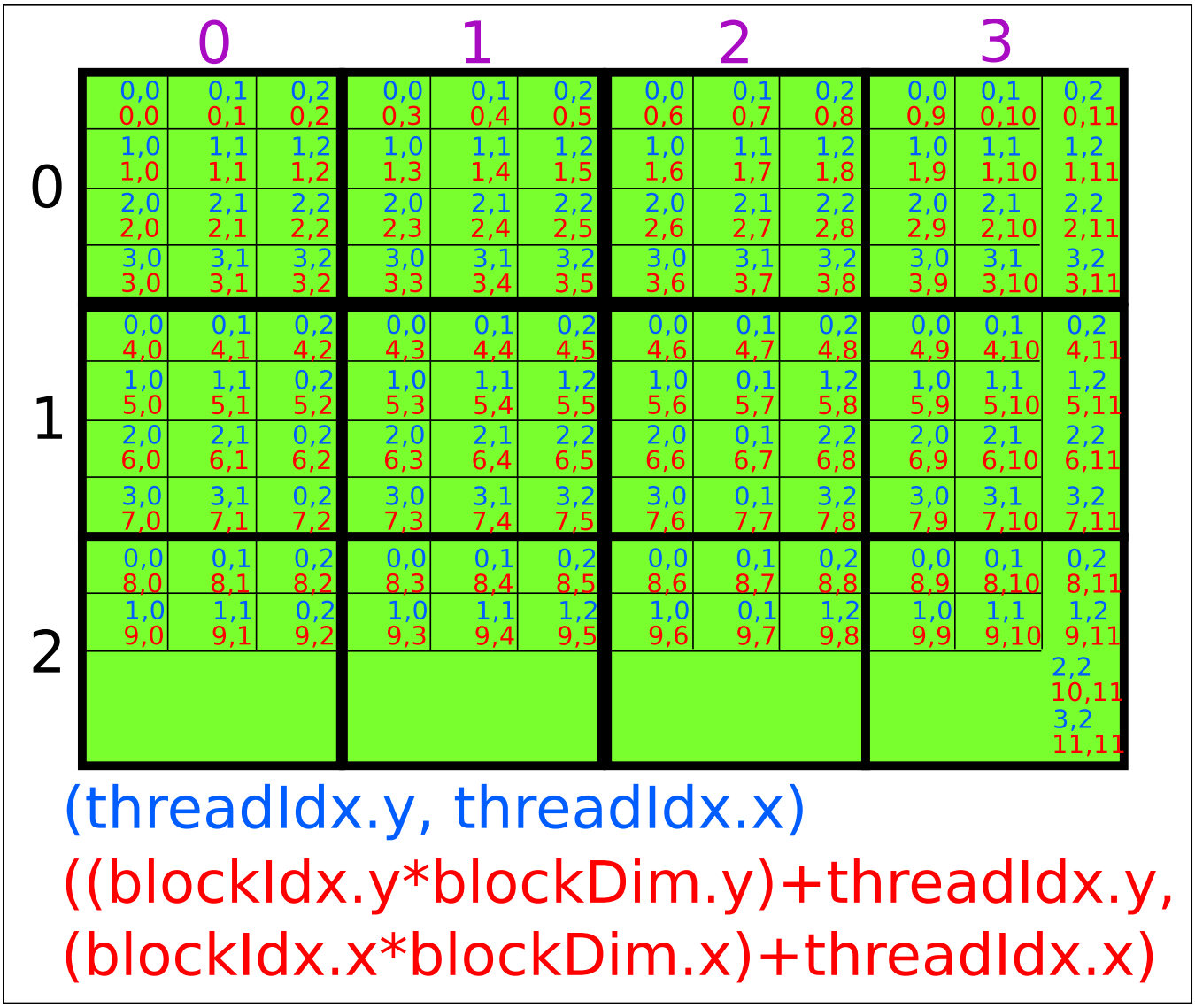
We must also break the threads up into blocks. A Cuda block can hold up to 1024 threads (typically), wich can hold all of our threads in this case. For the purposes of exposition, howeve, we will specify blocks to contain 3 threads in the X dimension and 4 threads in the Y dimension.

Note that matrices use [row][column] addressing and cuda uses [X][Y] addressing. Therefore:

$X = column$

$Y = row$

Thus, we can visually represent the thread IDs as follows:



Each matrix cell is labeled with its *threadIdx* values in blue, the actual matrix cell numbers in red. To compute the matix cell numbers, we need to know the block containing the thread, and the dimension of the blocks. In our example, we have:

blockDim.x=3

blockDim.y=4

So, the thread associated with matrix cell 7,6 (row 7, column 6), we have the following:

threadIdx.x = 0 (column)

threadIdx.y = 3 (row)

blockIdx.x = 2 (column)

blockIdx.y = 1 (row)

Cell address = *((blockIdx.y * blockDim.y)+threadIdx.y, (blockIdx.x * blockDim.x) + threadIdx.x) = (row, column)*

Cell address = *(7,6)*

Note: Note that this example equated the *X* values with columns and the *Y* values with rows. You can switch this if you like, just make sure to carry the switch through all the calculations.

Warning: Notice that the blocks extend past the end of the array. Threads WILL be created with IDs for these areas, and they will be accessing memory out of the array bounds. In each individual kernel, you must check to make sure the threadID corresponds to a matrix cell that is IN ARRAY BOUNDS! If not, your program will probalby crash.
If a Cuda program crashes within a kernel, it can be difficult to debug. A good rule of thumb is to first make sure you are not exceeding array bounds.

Now that we can determine thread IDs, we can construct a simple kernel. Remember, each thread will have 2 values fixed, and iterate over all remaining possibilities to form triples. The following code accomplishes this:

```
__global__ void sum3Kernel( int* device_nums, int arraySize, int* count )
{
    const int aIndex = threadIdx.x + (blockDim.x * blockIdx.x);
    const int bIndex = threadIdx.y + (blockDim.y * blockIdx.y);
    const int size = arraySize;

    if( aIndex < size-2 && bIndex < size-1 && bIndex > aIndex )
    {
        for( int i = bIndex+1; i < size; i++ )
        {
            if( device_nums[aIndex] + device_nums[bIndex] + device_nums[i ] == 0 )
            {
                atomicAdd( count, 1 );
            }
        }
    }
}
```

The *__global__* keyword indicates that the kernel can be called from code executing on both the CPU and GPU. The function takes 3 variables:

1. *device_nums*: An array of integers that we are computing the Sum3 algorithm upon.
2. *arraySize*: The length of the array *device_nums*.

3. *count*: A pointer to an integer allocated in GPU memory. Each time we find a triple that sums to 0, we will increment the count.

Lines 3 and 4 use the thread and block IDs to identify a pair of numbers in the *device_nums* array. We assume we will launch threads with 2 dimensional IDs. Each ID will form the index of a number in the *device_nums* array. The kernel will then form all possible triples with numbers from *device_nums* that use those two indexes.

Line 7 just makes sure that we are staying in the array bounds, and that we are not creating duplicate triples.

The loop on line 9 iterates over the appropriate values in the array to form triples.

Line 11 tests a triple to see if it sums to 0.

Line 13 increments the count variable. Since we may have a very large number of threads, and all threads are going to access the same location in GPU memory, a raw memory access will have race conditions. To get around this, we use an **atomic** function. The atomic function ensures that the increments to count will happen as if the program was a serial program (i.e., no race conditions will occur). This can negatively affect performance. In our case, not very many triples will end up summing to 0, so it shouldn't affect performance too much.

At this point, we have a kernel that will divide up work among the threads in a reasonable fashion. The rest of program is just setting up memory, and launching the right number of threads!

3.5.2 Step 2: Setting up GPU Memory

Our kernel takes three arguments. One is the *arraySize* integer. By inspecting the function, you should see that is a *pass by value* argument, meaning that the actual integer will copied into a variable in the local scope of the function. This is achieved in the normal way of using a *pass by value* variable in a function call. The other two arguments are pointers to memory addresses. Because the kernel is run on the GPU, these pointers **MUST** point to allocated memory on the GPU. So, we must allocate that memory.

The process to allocate memory on the GPU is the same as on the CPU. We simply use a memory allocation function to allocate the memory, then we initialize it. The only difference is that we use allocation functions provided by Cuda. Also, we can either initialize the memory using a kernel, or we can just copy data into the allocated memory from CPU memory. So, the usual recipe goes as follows:

1. Allocate memory on the CPU
2. Initialize the memory using code run on the CPU
3. Allocate an identical memory structure on the GPU
4. Copy the initialized memory from the CPU to the GPU

Again, we must simply use Cuda provided functions to do the GPU stuff. Here is the code to intialize the *device_nums* array and the *count* integer:

```
1  // Create an array of numbers in traditional memory.
2  // Fill the array with random values
3  std::vector<int> data;
4  srand(0);
5  for( int i = 0; i < total; i++ )
6  {
7      int tmp = myRandInt( ); // retuns a random integer between -100 and 100
8      data.push_back( tmp );
9  }
10
11 // Step 1:  Create an array on the GPU to hold the numbers that we
12 // will do the sum3 computation on
13 int *device_nums;
14 cudaMalloc( &device_nums, data.size() * sizeof( int) );
```

```
15
16 // Step 2: Copy the data to the device array
17 cudaMemcpy(device_nums, &(data[0]), data.size() * sizeof( int), cudaMemcpyHostToDevice);
18
19 //Step 3:  We must keep track of the number of triples that sum
20 // to 0.  We will create a single memory location (variable) on
21 // the GPU that is SHARED among ALL threads.  Whenever a thread
22 // finds a triple that sums to 0, this variable will be incremented
23 int* device_count;
24 cudaMalloc( &device_count, sizeof( int) );
25 {
26     // initialize the count to 0
27     int startCount = 0;
28     cudaMemcpy(device_count, &(startCount), sizeof( int), cudaMemcpyHostToDevice);
29 }
```

In the above code, the array is allocated first. Line 3 creates a vector that is then initialized with random numbers in lines 5-9.

To create the GPU memory, we first need a pointer to record the address of the memory we allocate (line 13). The GPU memory allocation function is *cudaMalloc* (line 14). We pass the pointer to hold the address of the allocated memory, and size of the chunk of memory we want to be allocated.

To copy data to the GPU, we simply use *cudaMemcpy* (line 17). It takes a pointer to the GPU memory, a pointe to the CPU memory to be copied, and the size of the chunk of memory to be copied. Finally, we pass a constant *cudaMemcpyHostToDevice*. This just is just a check to ensure that we are copying data form the host (CPU) to the device (GPU). Really, this just makes the code easier to read.

The *count* variable is allocated and initialized in exactly the same way (lines 23-29). The only difference is that a single integer is allocated instead of an array of integers.

Note: Only 1 integer is created and called *count*. This means ALL threads executing the kernel above access the SAME memory location. This is why the atomic function is required to avoid race conditions.

3.5.3 Step 3: Launching the Threads

Now that we have set up the kernel, allocated the memory, and intialized the memory, it is time to launch the threads. Recall that the kernel expects threads to be numbered with a 2-dimensional numbering scheme. In order to to this, we first have to decide **how many threads we want in each block**. Based on that concept, we can then do some arithmetic to determine how many blocks are required to generate threads over our entire data set. Cuda provides a struct with 3 integers that allow us to specify X, Y, and Z values for the numbers of threads per block and the number of blocks. It is called *dim3*.

So, to record the number of threads per block we want, we simply create and initialize a *dim3* struct. To indicate the number of blocks to create, we simply divide the threads per block into the array size along each dimension.

Remember, we want each thread to have a 2-dimensional ID, so we create threads in 2-dimensions. The thread IDs will index two spots in the array. Note that (take a look at the CPU code above), the thread IDs that we need only form an upper-triangular matrix of valid IDs (i=0, j=i+1, etc). This means about half of our threads won't actually do anything. But, thats OK! although not optimal. Fig. 3.7 shows the values of i and j that must be used to form triples. Clearly, we generate a lot of threads that do nothing. The good thing is, these threads will launch, run for a couple clock cycles (basically just compute their IDs and hit the *if* statement), and then be done. Overall, it is not a big deal in this case.

The code below has the code to assign the number of threads per block and the number of blocks to create based on array size. The kernel launch has a funny <<< >>> syntax. This is used to tell Cuda how many threads to launch,

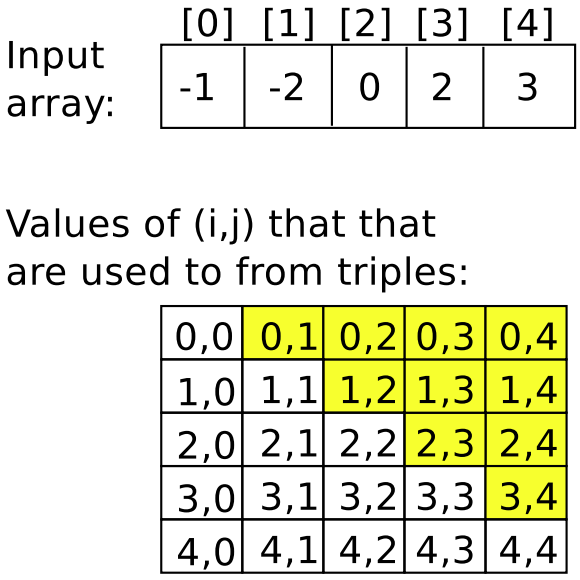


Fig. 3.7: A depiction of the values of i and j that must be used to form triples for testing in the sum3 problem. The shaded cells are the values of i and j that must be used.

and how to arrange them into blocks:

```
dim3 threadsPerBlock(16,32);
dim3 numBlocks((data.size() +threadsPerBlock.x-1)/ threadsPerBlock.x,

sum3Kernel<<< numBlocks, threadsPerBlock>>> ( device_nums, data.size(), device_count );
```

Many GPUs have a max block size of 1024 threads. Check the documentation of your GPU to be sure. Here, we use a smaller block size. Try playing around with block sizes to see how they affect performance.

3.5.4 Step 4: Transfer the result back to GPU memory

The final step is to simply transfer the vlaue of *count* from the GPU back to the CPU so we can print it out. This occurs with a *cudaMemcpy* just like transferring data to the GPU. The code is below:

```
int totalFound;
cudaMemcpy(&totalFound, device_count, sizeof( int), cudaMemcpyDeviceToHost);
```

3.5.5 All Together Now

The complete code is shown below.

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <vector>
#include <string>



/*****************************************************
 **********************
```

```
 This file is a cuda implementation of the sum3 program.

 It takes an array of numbers, and counts how many triples
 in the array sum to 0.  To do this, the program creates
 a 3 dimensional array of computations such that each cell
 in the array represents the computation of the sum of 3 numbers
 in the array.  There are errors in that duplicates are not accounted for,
 but the code works.

 **********************
 ******************************************************/



__global__ void sum3Kernel( int* device_nums, int arraySize, int* count )
{
    const int aIndex = threadIdx.x + (blockDim.x * blockIdx.x);
    const int bIndex = threadIdx.y + (blockDim.y * blockIdx.y);
    const int size = arraySize;

    if( aIndex < size-2 && bIndex < size-1 && bIndex > aIndex )
    {
        for( int i = bIndex+1; i < size; i++ )
        {
            if( device_nums[aIndex] + device_nums[bIndex] + device_nums[i ] == 0 )
            {
                atomicAdd( count, 1 );
            }
        }
    }
}

int myRandInt()
{
    int num = rand() % 100 +1;
    if( rand( ) % 2 == 0 )
        num *= -1;
    return num;
}

int main( int argc, char * argv[] )
{

    int total = 4;

    if( argc != 2 )
    {
        std::cerr << "usage: exe  [num of integers to generate for the sum3 computation]" << std::end
        exit( -1 );
    }
    {
        std::stringstream ss1;
        ss1 << argv[1];
        ss1 >> total;
    }

    // Create an array of numbers in traditional memory.
    // Fill the array with random values
```

```
    std::vector<int> data;
    srand(0);
    for( int i = 0; i < total; i++ )
    {
        int tmp = myRandInt( );
        data.push_back( tmp );
    }

    // Step 1:  Create an array on the GPU to hold the numbers that we
    // will do the sum3 computation on
    int *device_nums;
    cudaMalloc( &device_nums, data.size() * sizeof( int) );

    // Step 2: Copy the data to the device array
    cudaMemcpy(device_nums, &(data[0]), data.size() * sizeof( int), cudaMemcpyHostToDevice);

    //Step 3:  We must keep track of the number of triples that sum
    // to 0.  We will create a single memory location (variable) on
    // the GPU that is SHARED among ALL threads.  Whenever a thread
    // finds a triple that sums to 0, this variable will be incremented
    int* device_count;
    cudaMalloc( &device_count, sizeof( int) );
    {
        // initialize the count to 0
        int startCount = 0;
        cudaMemcpy(device_count, &(startCount), sizeof( int), cudaMemcpyHostToDevice);
    }


    // Just some code to time the kernel
    cudaEvent_t startTotal, stopTotal;
    float timeTotal;
    cudaEventCreate(&startTotal);
    cudaEventCreate(&stopTotal);
    cudaEventRecord( startTotal, 0 );



    // Step 4:  Decide how many threads we will organize into a block.  The
    // number of threads required will depend on the length of the array
    // containing random numbers.  Here, we are simply figuring out
    // how many threads we need based on the size of that array
    // (we allocated the array as an STL vector)
    //
    // Since EACH thread gets 2 fixed values, we are going to give threads
    // ID numbers that will indicate the array indexes of the 3 values
    // that will be fixed in that thread.  So, we create a 2 dimensional
    // thread block.  It simply labels each thread with 2 numbers that form
    // its identifier.
    dim3 threadsPerBlock(16,32);
    dim3 numBlocks((data.size() +threadsPerBlock.x-1)/ threadsPerBlock.x,
            (data.size() +threadsPerBlock.y-1)/ threadsPerBlock.y);

    std::cerr <<"data size: " <<(data.size()) << std::endl;
    std::cerr <<"block sizes: " <<(data.size() +threadsPerBlock.x-1)/ threadsPerBlock.x
        <<", " <<(data.size() +threadsPerBlock.y-1)/ threadsPerBlock.y <<  std::endl;


```

```
127     // Step 5.  Now we have computed how many threads to launch.  We have
128     // given each thread and identifier consisting of a pair (x,y).
129     // Finally, launch the threads.
130     sum3Kernel<<< numBlocks, threadsPerBlock>>> ( device_nums, data.size(), device_count );
131
132
133     // Step 6:  After the threads have all finished, the count of triples that
134     // sum to 0 is still stored on the GPU.  We just need to transfer it
135     // back to the CPU so we can print it out.
136     int totalFound;
137     cudaMemcpy(&totalFound, device_count, sizeof( int), cudaMemcpyDeviceToHost);
138
139     // stop the timer
140     cudaEventRecord( stopTotal, 0 );
141     cudaEventSynchronize( stopTotal );
142     cudaEventElapsedTime( &timeTotal, startTotal, stopTotal );
143     cudaEventDestroy( startTotal );
144     cudaEventDestroy( stopTotal );
145
146     // print it out!
147     std::cerr << "total time in seconds: " << timeTotal / 1000.0 << std::endl;
148     std::cerr << "Total triples found: " << totalFound  <<std::endl;
149 }
```

Exercise

Compile and run the sum3 Cuda program. See how long it takes to run on various inputs. Try different block sizes. Compile the CPU version of the code and compare running times.

3.5.6 Cuda Error Checking

One thing to remember is that the kernels are running on the GPU, which impacts how errors are reported. If you remember the first time you had a seg fault using C code, you may recall that it was initially baffling. The program crashed and really gave you no clues as to where in the code the error occurred. The nice thing about seg faults is that you can usually find out rather easily that the crash was indeed a seg fault. If you have a memory error on the GPU, it is even harder to tell what is going on. Basically, the program will continue to run, in many cases, but the GPU your GPU computations just won't work. If you are lucky, you will get some kind of message indicating that something went wrong.

When an error does occur on the GPU, error flags are set; one must simply check to see if any error flags are set after making a Cuda call. The following version of the code is identical to the code above, but has some macros and functions that will check the error flags on the GPU and halt the program if one is set. The code has been floating around the Internet for a while, and originated in the examples that come with the Cuda SDK. Here is the full Sum3 code with the error checking code. Do note that there is a comment about one of the error checking calls affecting performance (lines 60-68). If you are timing your code, you might want to comment that line out, or just turn off error checking by commenting out the *#define CUDA_CHECK_ERROR* line (line 29).

```
1 #include <iostream>
2 #include <sstream>
3 #include <fstream>
4 #include <vector>
5 #include <string>
6
7
8
```

```
/***************************************************
 ***********************
 This file is a cuda implementation of the sum3 program.

 It takes an array of numbers, and counts how many triples
 in the array sum to 0.  To do this, the program creates
 a 3 dimensional array of computations such that each cell
 in the array represents the computation of the sum of 3 numbers
 in the array.  There are errors in that duplicates are not accounted for,
 but the code works.

 Note the usage of CudaSafeCall() and CudaCheckError(). You should use these
 becuase cuda does not always fail in an obvious way without these functions.
 Once things are working, you can remove these functions to speed things up a bit.
 ***********************
 ***************************************************/


///////////////error  checking stufff
// Enable this for error checking
#define CUDA_CHECK_ERROR

#define CudaSafeCall( err )     __cudaSafeCall( err, __FILE__, __LINE__ )
#define CudaCheckError()        __cudaCheckError( __FILE__, __LINE__ )

inline void __cudaSafeCall( cudaError err, const char *file, const int line )
{
#ifdef CUDA_CHECK_ERROR

    if ( cudaSuccess != err )
    {
        fprintf( stderr, "cudaSafeCall() failed at %s:%i : %s\n",
                file, line, cudaGetErrorString( err ) );
        exit( -1 );
    }
#endif  // CUDA_CHECK_ERROR

    return;
}

inline void __cudaCheckError( const char *file, const int line )
{
#ifdef CUDA_CHECK_ERROR
    cudaError_t err = cudaGetLastError();
    if ( cudaSuccess != err )
    {
        fprintf( stderr, "cudaCheckError() failed at %s:%i : %s.\n",
                file, line, cudaGetErrorString( err ) );
        exit( -1 );
    }

    // More careful checking. However, this will affect performance.
    // Comment if not needed.
    err = cudaDeviceSynchronize();
    if( cudaSuccess != err )
    {
        fprintf( stderr, "cudaCheckError() with sync failed at %s:%i : %s.\n",
                file, line, cudaGetErrorString( err ) );
```

```
        exit( -1 );
    }
#endif // CUDA_CHECK_ERROR

    return;
}

////////////////////end of error checking stuff


__global__ void sum3Kernel( int* device_nums, int arraySize, int* count )
{
    const int aIndex = threadIdx.x + (blockDim.x * blockIdx.x);
    const int bIndex = threadIdx.y + (blockDim.y * blockIdx.y);
    const int size = arraySize;

    if( aIndex < size-2 && bIndex < size-1 && bIndex > aIndex )
    {
        for( int i = bIndex+1; i < size; i++ )
        {
            if( device_nums[aIndex] + device_nums[bIndex] + device_nums[i ] == 0 )
            {
                atomicAdd( count, 1 );
            }
        }
    }
}

int myRandInt()
{
    int num = rand() % 100 +1;
    if( rand( ) % 2 == 0 )
        num *= -1;
    return num;
}

int main( int argc, char * argv[] )
{

    int total = 4;

    if( argc != 2 )
    {
        std::cerr << "usage: exe  [num of integers to generate for the sum3 computation]" << std::endl;
        exit( -1 );
    }
    {
        std::stringstream ss1;
        ss1 << argv[1];
        ss1 >> total;
    }

    // Create an array of numbers in traditional memory.
    // Fill the array with random values
    std::vector<int> data;
    srand(0);
    for( int i = 0; i < total; i++ )
    {
```

```
        int tmp = myRandInt( );
        data.push_back( tmp );
    }

    // Step 1:  Create an array on the GPU to hold the numbers that we
    // will do the sum3 computation on
    int *device_nums;
    CudaSafeCall( cudaMalloc( &device_nums, data.size() * sizeof( int) ) );

    // Step 2: Copy the data to the device array
    cudaMemcpy(device_nums, &(data[0]), data.size() * sizeof( int), cudaMemcpyHostToDevice);
    CudaCheckError();

    //Step 3:  We must keep track of the number of triples that sum
    // to 0.  We will create a single memory location (variable) on
    // the GPU that is SHARED among ALL threads.  Whenever a thread
    // finds a triple that sums to 0, this variable will be incremented
    int* device_count;
    CudaSafeCall( cudaMalloc( &device_count, sizeof( int) ) );
    {
        // initialize the count to 0
        int startCount = 0;
        cudaMemcpy(device_count, &(startCount), sizeof( int), cudaMemcpyHostToDevice);
        CudaCheckError();
    }


    // Just some code to time the kernel
    cudaEvent_t startTotal, stopTotal;
    float timeTotal;
    cudaEventCreate(&startTotal);
    cudaEventCreate(&stopTotal);
    cudaEventRecord( startTotal, 0 );



    // Step 4:  Decide how many threads we will organize into a block.  The
    // number of threads required will depend on the length of the array
    // containing random numbers.  Here, we are simply figuring out
    // how many threads we need based on the size of that array
    // (we allocated the array as an STL vector)
    //
    // Since EACH thread gets 2 fixed values, we are going to give threads
    // ID numbers that will indicate the array indexes of the 3 values
    // that will be fixed in that thread.  So, we create a 2 dimensional
    // thread block.  It simply labels each thread with 2 numbers that form
    // its identifier.
    dim3 threadsPerBlock(16,32);
    dim3 numBlocks((data.size() +threadsPerBlock.x-1)/ threadsPerBlock.x,
            (data.size() +threadsPerBlock.y-1)/ threadsPerBlock.y);

    std::cerr <<"data size: " <<(data.size()) << std::endl;
    std::cerr <<"block sizes: " <<(data.size() +threadsPerBlock.x-1)/ threadsPerBlock.x
        <<", " <<(data.size() +threadsPerBlock.y-1)/ threadsPerBlock.y <<  std::endl;


    // Step 5.  Now we have computed how many threads to launch.  We have
    // given each thread and identifier consisting of a pair (x,y).
```

```
    // Finally, launch the threads.
    sum3Kernel<<< numBlocks, threadsPerBlock>>> ( device_nums, data.size(), device_count );
    CudaCheckError();


    // Step 6:  After the threads have all finished, the count of triples that
    // sum to 0 is still stored on the GPU.  We just need to transfer it
    // back to the CPU so we can print it out.
    int totalFound;
    cudaMemcpy(&totalFound, device_count, sizeof( int), cudaMemcpyDeviceToHost);

    // stop the timer
    cudaEventRecord( stopTotal, 0 );
    cudaEventSynchronize( stopTotal );
    cudaEventElapsedTime( &timeTotal, startTotal, stopTotal );
    cudaEventDestroy( startTotal );
    cudaEventDestroy( stopTotal );

    // print it out!
    std::cerr << "total time in seconds: " << timeTotal / 1000.0 << std::endl;
    std::cerr << "Total triples found: " << totalFound  <<std::endl;
}
```

3.5.7 Performance

Now that we have the full example, we can see how they perform. The following table lists the execution times of the CPU version of the program without compiler optimizations, the CPU version with -O3 compiler optimizations, and the GPU version of the code. The time reported is the time for only the computation portion of the code, not setting up the arrays and generating random numbers. The CPU is an Intel Xeon. The GPU is a Tesla K40.

Array size	CPU	CPU -O3	GPU
800	1.05393	0.1131	0.0047007
1000	2.05775	0.219797	0.00916141
1200	3.55459	0.378601	0.015605
1400	5.64653	0.600016	0.024847
1600	8.42537	0.893907	0.0369319
1800	12.0064	1.27085	0.0521852
2000	16.4582	1.74128	0.0722541

The GPU gets a speedup of 227 times over the non-optimized CPU code, and a speedup of 24 times over the compiler optimize CPU code. Thats pretty good. Now, we need to make the GPU code even faster!

3.6 Optimizing

Section Goals:

1. Describe the memory hierarchy of the GPU.
2. Use shared memory to optimize kernel.

At this point, we have a working Sum3 program that uses the GPU, and it is pretty fast. The last bit is just to optimize. We are following the usual path of getting something working first, then worrying about optimizations. This is usually a good plan to follow, since debugging a non-optimized implementation tends to be much easier.

When it comes to optimizing Cuda programs, one tends to focus on two venues:

1. Reducing thread divergence.
2. Optimizing memory access.

Memory optimizations, in particular, can lead to an order of magnitude improvement in performance!

3.6.1 Reducing Thread Divergence

The case for reducing thread divergence is obvious. If two threads in a block diverge, one must simply stall. So, its best to try to arrange our blocks such that threads within a block don't diverge, or at least minimize divergence. Blocks are used a unit of grouping threads into groups of functional units on SMX cores.

3.6.2 Optimizing Memory: Global Memory Accesses

Optimizing memory access is the other major focus. This tends to follow two avenues, the first is optimizing **global memroy accesses**.

Global Memory

> Global Memory is just the term for main memory on the GPU. When you use *cudaMalloc*, you are allocating memory in the GPU's *global memory*.

Remember, the GPU is running many threads at once. Thus, in any clock cycle, many memory requests can be issued at once. GPU memory is highly banked, and these memory requests are handled most efficiently when threads are requesting memory stored in different memory banks. That way, each bank can find the requested memory in parallel.

In the GPU, memory tends to be allocated in arrays. One simple practice that a programmer can do to encourage efficient global memory access is to structure the block dimensions so that groups of threads that need to fetch the same piece of memory are grouped into the same block! Thus, block dimensions can have a big impact on memory access performance! If multiple threads need the same piece of memory, and all generate a memory request in the same clock cycle, then the GPU memory system only needs to handle 1 memory access. In essence, those memory accesses are *coalesced* into a single memory operation.

Another simple practice it to arrange data in multiple arrays, rather than an array of structs. For example, consider a program that is computing values based on points consisting of an (x, Y) coordinate. The traditional way to manage this memory is to put the (x, y) values into a struct, then create an array of structs. However, if a thread needs to get both X and Y from memory for a particular point, they may be in the same memory bank, requiring 2 memory requests to the same bank that must be serialized. This is an overly simplistic example, but it gets the idea across. If instead, the programmer creates two parallel arrays, one containing X values and the other containing Y values, then X and Y will be stored far apart in memory, and will likely be stored on different memory banks.

3.6.3 Optimizing Using Shared Memory:

Again, global memory is the main memory on the GPU. The GPU does do some caching, but it is not as effective as a traditional memory hierarchy with multiple levels of cache, mainly due to the sheer volume of data being accessed by thousands of threads. Thus, a completely automatic caching system is not appropriate. Instead, a small, programmer controlled, high speed memory exists on processor called **shared memory**.

Shared Memory

> Shared memory is a small amount of memory, similar to a cache, that is on processor. It is different from cache in that it is controlled directly by the programmer. Shared memory is accessible to all threads grouped into a *block*.

In particular, there are 5 main things to remember about shared memory:

1. Because it is programmer controlled, shared memory must be *explicitly* allocated in a kernel.

2. Shared memory is allocated for each *thread block*. This means that all threads in a block can access the same shared memory locations. But a thread from one block cannot directly access the shared memory from another block.
3. Shared memory is essentially local in scope to the block in which it is defined. When all threads in a particular block terminate, the shared memory associated with that block is automatically deallocated.
4. Unlike cache, the programmer must write the values of shared memory locations back to global memory locations if those values need to exist beyond the lifetime of a thread block.
5. Shared memory is often used like cache: to take advantage of temporal locality of memory references and not have to wait on the slower, global memory access.

So, lets take a look at our sum3 kernel. Each kernel computes its thread ID number, and uses that number to index the array of random numbers and retrieve 2 values from global memory. Those two values are used over and over again. They might be good candidates for shared memory, but since they don't change, its probably better just to assign them to constant local variables. Doing this will indicate to the compiler that those values should be stored in registers in the processor, instead of repeatedly accessing global memory to find them. This first optimization looks like the following:

```
__global__ void sum3Kernel( int* device_nums, int arraySize, int* count )
{
    const int aIndex = threadIdx.x + (blockDim.x * blockIdx.x);
    const int bIndex = threadIdx.y + (blockDim.y * blockIdx.y);
    const int size = arraySize;

    if( aIndex < size-2 && bIndex < size-1 && bIndex > aIndex )
    {
        const int item1 = device_nums[aIndex];
        const int item2 = device_nums[bIndex];
        for( int i = bIndex+1; i < size; i++ )
        {
            if( item1 + item2 + device_nums[i] == 0 )
            {
                atomicAdd( count, 1 );
            }
        }
    }
}
```

It turns out, that when we make this optimization, the running times stay exactly the same. This is probably due to a few reasons:

1. Optimizing compilers are really good, and probably did this already at the machine code level.
2. The biggest slowdown is probably the atomic function, as threads must effectively serialize on atomic memory accesses.

So, now lets use some shared memory. The biggest slowdown is probably the atomic function, so lets deal with that. Really, each thread only needs to keep track of its own count. Then, we can combine counts at the end. Using shared memory will allow each thread to keep track of its own count. Then, each block can sum up the count for that block. Finally, we can use a single atomic call at the end that just reports an entire block's count back to global memory using the atomic function. This will greatly reduce the number of atomic function calls.

So, step 1 is to allocate enough shared memory integers so that each thread can have its own, private shared memory location where it can keep its *count*. Remember, that thread numbers in a block always start with 0. The easiest way to do this is to just create an array of integers in shared memory for each block, and have each thread access only 1 of the values in that array (based on its thread ID). In our case, we need a 2 dimensional array as follows:

```
__shared__ int countArray[blockSizeRows][blockSizeCols];
```

The above line creates an array with the same dimensions as the thread block, so that each thread can index 1 array element based on its ID. Note the *__shared__* keyword that indicates this is shared memory.

Now, each thread simply increments its count variable as an array location in *countArray*. Again, a thread access only the location corresponding to its thread ID (using 2-dimensional thread IDs).

```
const int item1 = device_nums[aIndex];
const int item2 = device_nums[bIndex];
for( int i = bIndex+1; i < size; i++ )
   if( item1 + item2 + device_nums[ i ] == 0 )
   {
       countArray[threadIdx.x][threadIdx.y]++;
   }
```

Once that is finished, each element in the shared array has a count corresponding to a single thread. We must now sum those numbers to determine the total count for the thread block. The easiest way to do this is to just have 1 thread compute that sum. This is where branch diversion will actually help us. The first step to make sure ALL threads in block have finished summing their triples. Not all threads in a block will finish at the exact same time due to scheduling issues within the processor. This can be achieved using a **barrer synchronization** construct:

Barrier synchronization:

> A thread synchronization construct in which no thread is allowed to pass a specified barrier (usually a certain line of code) until **all** other threads have reached that barrier.

In Cuda, block-level barrier synchronization exists using the function call *__synchthreads();*. If this function is in a kernel, no thread in a **block** will be able to pass that line of code until all other threads in that **block** reach it. This gives us an easy mechanism to make sure that all threads in a block reach a certain point in a kernel before letting any more work take place.

> **Warning:** *__synchthreads();* must be on a path of execution such that ALL threads in a block actually execute it. In other words, you should be very careful (and probably just avoid, in most cases) putting *__synchthreads();* in the body of an *if* block or in the body of a loop.
> *__synchthreads();* only provides barrier synchronization on a **per block** basis. All threads in a block conform to the synchronization. Threads in different blocks will **not** be synchronized.

In our kernel, we want to wait until all threads are done creating their own counts, then we will sum up the counts for the entire block. The easiest way to sum the counts for the block is to just pick 1 thread to do it. Thus, we will wrap the sum code in an *if* statement that will only allow the thread with ID *(0,0)* enter it. All other threads will diverge, so **only** thread *(0,0)* computes the sum. Once the sum is computed, we will perform a single atomic call to the shared memory location *count*. Thus, instead of having each thread possible serialize on a bunch of atomic calls, we will have only 1 atomic call per *block* of threads. The complete kernel is as follows:

```
const int blockSizeRows = 16;
const int blockSizeCols = 32;

__global__ void sum3Kernel( int* device_nums, int arraySize, int* count )
{

        __shared__ int countArray[blockSizeRows][blockSizeCols];
        int total;
        const int aIndex = threadIdx.x + (blockDim.x * blockIdx.x);
        const int bIndex = threadIdx.y + (blockDim.y * blockIdx.y);
        const int size = arraySize;

        countArray[threadIdx.x][threadIdx.y] = 0;
```

```
        __syncthreads();

        if( aIndex < size && bIndex < size && bIndex > aIndex )
        {
            const int item1 = device_nums[aIndex];
            const int item2 = device_nums[bIndex];
                for( int i = bIndex+1; i < size; i++ )
                        if( item1 + item2 + device_nums[ i ] == 0 )
                        {
                                countArray[threadIdx.x][threadIdx.y]++;
                        }
        }
        __syncthreads();
        if( threadIdx.x == 0 && threadIdx.y == 0 )
        {
                total = 0;
                for( int i = 0; i < blockDim.x; i++ )
                        for( int j = 0; j < blockDim.y; j++ )
                                total += countArray[i][j];
                if( total > 0 )
                {
                        atomicAdd( count, total );
                }
        }

}
```

Finally, the complete code is shown as follows:

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <vector>
#include <string>



/*****************************************************
**********************
        This file is a cuda implementation of the sum3 program.

        It takes an array of numbers, and counts how many triples
        in the array sum to 0.  To do this, the program creates
        a 3 dimensional array of computations such that each cell
        in the array represents the computation of the sum of 3 numbers
        in the array.  There are errors in that duplicates are not accounted for,
        but the code works.

        Note the usage of CudaSafeCall() and CudaCheckError(). You should use these
        becuase cuda does not always fail in an obvious way without these functions.
        Once things are working, you can remove these functions to speed things up a bit.
**********************
*****************************************************/


///////////////error  checking stufff
// Enable this for error checking
//#define CUDA_CHECK_ERROR

```

```
#define CudaSafeCall( err )     __cudaSafeCall( err, __FILE__, __LINE__ )
#define CudaCheckError()        __cudaCheckError( __FILE__, __LINE__ )

inline void __cudaSafeCall( cudaError err, const char *file, const int line )
{
#ifdef CUDA_CHECK_ERROR

    if ( cudaSuccess != err )
    {
        fprintf( stderr, "cudaSafeCall() failed at %s:%i : %s\n",
                file, line, cudaGetErrorString( err ) );
        exit( -1 );
    }
#endif  // CUDA_CHECK_ERROR

    return;
}

inline void __cudaCheckError( const char *file, const int line )
{
#ifdef CUDA_CHECK_ERROR
    cudaError_t err = cudaGetLastError();
    if ( cudaSuccess != err )
    {
        fprintf( stderr, "cudaCheckError() failed at %s:%i : %s.\n",
                file, line, cudaGetErrorString( err ) );
        exit( -1 );
    }

    // More careful checking. However, this will affect performance.
    // Comment if not needed.
    err = cudaDeviceSynchronize();
    if( cudaSuccess != err )
    {
        fprintf( stderr, "cudaCheckError() with sync failed at %s:%i : %s.\n",
                file, line, cudaGetErrorString( err ) );
        exit( -1 );
    }
#endif // CUDA_CHECK_ERROR

    return;
}
//////////////////end of error checking stuff

const int blockSizeRows = 16;
const int blockSizeCols = 32;

__global__ void sum3Kernel( int* device_nums, int arraySize, int* count )
{

        __shared__ int countArray[blockSizeRows][blockSizeCols];
        int total;
        const int aIndex = threadIdx.x + (blockDim.x * blockIdx.x);
        const int bIndex = threadIdx.y + (blockDim.y * blockIdx.y);
        const int size = arraySize;

        countArray[threadIdx.x][threadIdx.y] = 0;
        __syncthreads();
```

```
89
90          if( aIndex < size && bIndex < size && bIndex > aIndex )
91          {
92              const int item1 = device_nums[aIndex];
93              const int item2 = device_nums[bIndex];
94                  for( int i = bIndex+1; i < size; i++ )
95                          if( item1 + item2 + device_nums[ i ] == 0 )
96                          {
97                                  countArray[threadIdx.x][threadIdx.y]++;
98                          }
99          }
100         __syncthreads();
101         if( threadIdx.x == 0 && threadIdx.y == 0 )
102         {
103                 total = 0;
104                 for( int i = 0; i < blockDim.x; i++ )
105                         for( int j = 0; j < blockDim.y; j++ )
106                                 total += countArray[i][j];
107                 if( total > 0 )
108                 {
109                         atomicAdd( count, total );
110                 }
111         }
112
113 }
114
115 int myRandInt()
116 {
117         int num = rand() % 100 +1;
118         if( rand( ) % 2 == 0 )
119                 num *= -1;
120         return num;
121 }
122
123 int main( int argc, char * argv[] )
124 {
125
126         int total = 4;
127
128         if( argc != 2 )
129         {
130                 std::cerr << "usage: exe  [num nof nums]" << std::endl;
131                 exit( -1 );
132         }
133         {
134                 std::stringstream ss1;
135                 ss1 << argv[1];
136                 ss1 >> total;
137         }
138
139         // create a data vec
140
141         std::vector<int> data;
142
143         for( int i = 0; i < total; i++ )
144         {
145                 int tmp = myRandInt( );
146                 data.push_back( tmp );
```

```
                //std::cerr<< tmp << std::endl;
        }
        //cudaSetDevice(0);
        // create a device vector
        int *device_nums;
        CudaSafeCall( cudaMalloc( &device_nums, data.size() * sizeof( int) ) );
        // copy the data to the device
        cudaMemcpy(device_nums, &(data[0]), data.size() * sizeof( int), cudaMemcpyHostToDevice);
        CudaCheckError();


        // put the count on the device
        int* device_count;
        CudaSafeCall( cudaMalloc( &device_count, sizeof( int) ) );
        {
                int startCount = 0;
                cudaMemcpy(device_count, &(startCount), sizeof( int), cudaMemcpyHostToDevice);
                CudaCheckError();
        }


        //Timing code
        cudaEvent_t startTotal, stopTotal;
        float timeTotal;
        cudaEventCreate(&startTotal);
        cudaEventCreate(&stopTotal);
        cudaEventRecord( startTotal, 0 );



        //create the kernel
        dim3 threadsPerBlock(blockSizeRows,blockSizeCols);
        dim3 numBlocks( (data.size() +threadsPerBlock.x-1)/ threadsPerBlock.x,
                                        (data.size() +threadsPerBlock.y-1)/ threadsPerBlock.y );

        std::cerr <<"data size: " <<(data.size()) << std::endl;
        std::cerr <<"block sizes: " <<(data.size() +threadsPerBlock.x-1)/ threadsPerBlock.x
                  <<", " <<(data.size() +threadsPerBlock.y-1)/ threadsPerBlock.y <<  std::endl;

        sum3Kernel<<< numBlocks, threadsPerBlock>>> ( device_nums, total, device_count );

        CudaCheckError();


        // finally, get the count off the GPU
        int totalFound;
        cudaMemcpy(&totalFound, device_count, sizeof( int), cudaMemcpyDeviceToHost);


        cudaEventRecord( stopTotal, 0 );
        cudaEventSynchronize( stopTotal );
        cudaEventElapsedTime( &timeTotal, startTotal, stopTotal );
        cudaEventDestroy( startTotal );
        cudaEventDestroy( stopTotal );
        std::cerr << "total time in seconds: " << timeTotal / 1000.0 << std::endl;
        std::cerr << "Total triples found: " << totalFound  <<std::endl;
}
```

3.6.4 Performance

The following table depicts the running times of the C, optimized C, Cuda, and shared memory Cuda versions of the sum3 program (in seconds). The shared memory version of the program runs in about half the time of the first GPU program:

Array Size	CPU	CPU -O3	GPU	GPU Shared Memory
800	1.05393	0.1131	0.0047007	0.00263213
1000	2.05775	0.219797	0.00916141	0.00477453
1200	3.55459	0.378601	0.015605	0.00793862
1400	5.64653	0.600016	0.024847	0.0121216
1600	8.42537	0.893907	0.0369319	0.0177161
1800	12.0064	1.27085	0.0521852	0.0249915
2000	16.4582	1.74128	0.0722541	0.0336659

- search

[GFS2003] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM, New York, NY, USA, 29-43. DOI=10.1145/945445.945450 http://doi.acm.org/10.1145/945445.945450

[GOM1995CG] 1. Gajentaan, M.H. Overmars, "On a class of $O(n2)$ problems in computational geometry", Computational Geometry: Theory and Applications, 1995, 5 (3): 165–185, doi:10.1016/0925-7721(95)00022-2.

[OpenMPDoc] http://openmp.org/

[OpenMPWiki] https://en.wikipedia.org/wiki/OpenMP